HAW HAMBURG

BACHELORTHESIS
Simon Borho

Qualitätssicherung in der Android App-Entwicklung durch Testautomatisierung

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Simon Borho

Qualitätssicherung in der Android App-Entwicklung durch Testautomatisierung

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

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr.-Ing Marina Tropmann-Frick

Eingereicht am: 6. Mai 2021

Simon Borho

Thema der Arbeit

Qualitätssicherung in der Android App-Entwicklung durch Testautomatisierung

Stichworte

Android, Testautomatisierung, Modultests, Integrationstests, Tests der Benutzeroberfläche, Kontinuierliche Integratio

Kurzzusammenfassung

Bei der Entwicklung von Android Apps gibt es einige Arten die Software und die dazugehörigen Systeme zu testen. Die Herausforderung besteht darin, die passenden Tests der Benutzeroberfläche, der Integration sowie der Module zu entwickeln, um die Anwendung bestmöglich zu überprüfen. Hierzu soll ein Testaufbau entwickelt werden. Dieser soll eine Umgebung für die Tests der Anwendung bereitstellen und mit diesen die Qualität der Anwendung langfristig anheben. Um sich bestmöglich darauf vorzubereiten, sollen in dieser Arbeit einige der meist verwendeten Praktiken und Vorgehensweisen beleuchtet und exemplarisch angewendet werden. Um den Prozess der Entwicklung und Auslieferung weitergehend zu optimieren, soll ein exemplarischer Aufbau eines System der kontinuierliche Integration beschrieben werden.

Simon Borho

Title of Thesis

Quality assurance in Android app development through test automation

Keywords

Android, test automation, Unit Tests, Integration Tests, UI Tests, continuous integration

Abstract

When developing Android apps there are several ways to test the software and associated systems. The challenge is to develop suitable UI, integration and unit tests to test an

application in the best possible way. For this purpose a test setup is to be developed. This is intended to provide an environment for testing the application and using these to raise the quality of the application in the long term. To prepare for this this thesis will highlight and exemplify some of the most commonly used practices and approaches. To further optimize the process of development and delivery an exemplary setup of a continuous integration system will be described.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Diese Arbeit beschäftigt sich mit der Thematik der Testautomatisierung für Android Anwendungen, wie diese exemplarisch umgesetzt und das Nutzerverhalten simuliert werden kann. In der Einleitung wird zunächst ein Einblick in das Themenfeld gegeben und welche Motivation hinter der Arbeit steckt. Zudem wird der rote Faden vorgestellt, welcher die Beschreibung der Kapitel enthält.

1.1 Motivation

Wie wichtig das Testen der erstellten Software ist, wird in nahezu jedem Kurs, Fach oder Fortbildung, welche sich mit der Thematik von Softwareentwicklung befasst, deutlich gemacht. Durch dies kann die Qualität der Software angehoben werden, was wiederum zu weniger Abstürzen und Fehlern führt und somit die Zufriedenheit der Kunden und Nutzer anhebt. Software sollte daher vor der Auslieferung immer getestet werden. Hierfür gibt es mehrere Möglichkeiten. Eine davon besteht darin, dass der Entwickler, Kunde oder geschultes Personal durch die Software navigiert und dabei alle Anwendungsfälle abdeckt. Ebenso wird darauf geachtet, dass auf der Bedienoberfläche Abstände, Farben und Formen korrekt dargestellt werden. Dies ist jedoch auf längere Sicht eine sehr aufwändige und somit kostenintensive Vorgehensweise, welche meist eine Vernachlässigung der Tests mit sich bringt.[40] Eine auf lange Laufzeit gesehene, praktikablere und effizientere Methode ist das Testen durch automatisierte Tests. Hierbei wird der Code von seinem kleinsten Bestandteil, den einzelnen Funktionen, bis hin zur ausführlichen Interaktion und Optik der Oberflächen getestet. Ebenso können ganze Szenarien durchgeführt werden, welche zu Beginn der Entwicklung oder Updates eines Projekts als User Stories definiert werden. Die Motivation hinter dieser Arbeit besteht darin, tiefere Einblicke in die Praxis der zweiten Methodik zu erhalten und diese gegebenenfalls auch in der Produktion anzuwenden.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist es, eine in Kotlin geschriebene Anwendung für mobile Android Geräte mit Hilfe von automatisierten Tests in Teilen zu prüfen. Um dies umzusetzen soll zunächst erläutert werden worin die Vorteile liegen und welche Grundlagen man benötigt. Anschließend soll die Theorie und Praxis von Modultests für Komponenten beschrieben und angewandt werden. Nachfolgend werden die Komponenten zusammengeführt und ein Integrationstests durchgeführt. Hierbei wird überprüft, ob die Integration der Module miteinander wie erwartet funktioniert. Im letzten Schritt sollen Tests der Benutzeroberfläche durchgeführt werden. Sie dienen zum Validieren und Simulieren zusammenhängender Aktionen eines Benutzers und testen das gesamtheitliche System.[48] Um Modul- und Integrationstests unabhängig ausführen zu können, ist es zudem vonnöten ein Framework für das Erzeugen von Test-Doubles zu verwenden. Die hierbei entwickelten Tests dienen zur Veranschaulichung der jeweiligen Testart und sollen zeigen, ob diese mit denen in der Praxis verwendeten Praktiken kompatibel ist. Des Weiteren soll ein System der kontinuierlichen Integration erläutert werden. Hierdurch kann der Prozess des Testens, nach der Übergebe des Codes in ein Versionsverwaltungsprogramm, automatisch gestartet werden.[2]

1.3 Aufbau der Arbeit

Der Aufbau der Arbeit gliedert sich in die nachfolgend aufgeführten Kapitel.

Grundlagen

In diesem Kapitel wird auf die Wirtschaftlichkeit von Testautomatisierung, die Unterschiede beim Testen zwischen Emulatoren und physischen Geräten und dem Stand der Technik im Bezug auf das Testen erläutert. Zudem wird das Konzept hinter Mockups erläutert und mit MockK veranschaulicht.

Modultests

Das dritte Kapitel dreht sich um die Grundlagen der Modultests, welche Vorteile diese bieten und wie sie in einem Android-Projekt umgesetzt werden können.

Integrationstest

Das Kapitel beinhaltet die Grundlagen der Integrationstests, wie diese für ein Fragment umgesetzt werden und welche Rahmenstrukturen verwendet werden können.

Tests der Benutzeroberfläche

Im vierten Kapitel geht es um das Testen der Benutzeroberfläche und welche Möglichkeiten es gibt, eine User Story zu testen. Zudem wird das Werkzeug Android Monkeyrunner beleuchtet, welches selbständig und zufällig durch eine Anwendung navigiert.

Kontinuierliche Integration

In diesem Kapitel wird der theoretische Aufbau eines Servers zur kontinuierlichen Integration erläutert und welche Vorteile ein solches System mit sich bringt.

Fazit und Ausblick

Zum Schluss der Arbeit wird ein Fazit über die zuvor beschriebenen und gezeigten Herangehensweisen gezogen, welche Methoden oder Herangehensweisen in der Produktion eventuell übernommen werden und wie das System zum automatisierten Testen von Android-Projekten ausgebaut werden kann.

2 Grundlagen

Dieses Kapitel widmet sich den Grundlagen des Testens. Hierzu wird auf die Fragen eingegangen, ob sich das automatisierte Testen aus wirtschaftlicher Sicht für ein Unternehmen lohnen kann und ob dabei ein Emulator oder ein physisches Gerät verwendet werden sollte. Zudem wird ein Blick auf den Stand der Technik in Bezug auf das Testen geworfen und die Grundlagen von Mockups beleuchtet.

2.1 Wirtschaftlichkeit

Sollte ein Unternehmen sich dazu entscheiden von manuellem auf automatisiertes Testen umzusteigen, werden zu Beginn einige Kosten auf das Unternehmen zukommen. So müssem Mitarbeiter intensiv in die neue Materie eingearbeitet werden. Zudem kann es vonnöten sein, dass neue Tools, Software und Hardware angeschafft werden müssen. Auf lange Sicht gesehen können die Gesamtkosten jedoch erheblich gesenkt werden, da die Anzahl der benötigten Personen und der Zeitaufwand, im Vergleich zum manuellen Testen, aufgrund der Wiederholbarkeit gesenkt werden kann.

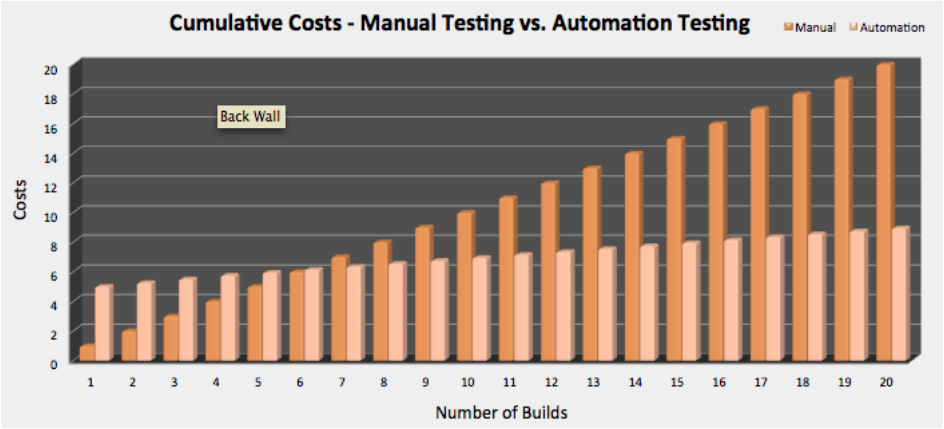


Abbildung 2.1: Kosten automatisierter vs manueller Tests

Der wahre Nutzen der Testautomatisierung liegt in der Wiederholbarkeit der Tests, aber auch in der Testausführung, welche meist nicht manuell durchgeführt wird. Wie in der Abbildung 2.1 zu sehen, steigen bei einer durchschnittlichen Anwendung, welche automatisiert getestet wird, zu Beginn die Kosten auf ca. das Fünffache im Vergleich zum manuellen Testen. Bereits ab dem siebten Build überschreiten die Kosten des manuellen Testens die des automatisierten Testens, welche von Build zu Build durch die Erweiterung und Anpassung der bestehenden Tests nur minimal ansteigen. Im Vergleich bleiben die Kosten für das manuelle Testen bei jedem Build gleich und addieren sich somit auf. Dies führt dazu, dass die Kosten für das manuelle Testen bereits nach dem 20. Build die des automatisierten Testens um das doppelte überschreiten.[50]

Für ein Unternehmen lohnt es sich aus finanzieller Sicht eine gut und ausführlich getestete Software auszuliefern. Ist dies nicht der Fall, können die Kosten bis zum Finden und der Beheben eines Fehlers exponentiell ansteigen. Kennt Beck schreibt dazu:

„Die meisten Fehler kosten am Ende mehr, als die Vermeidung gekostet hätte. Auftretende Fehler sind teuer, da nicht nur die direkten Kosten für die Behebung des Fehlers zu tragen sind, sondern auch die indirekten Kosten aufgrund von beschädigten Beziehungen, verlorenen Geschäften und verlorener Entwicklungszeit.“ [34]

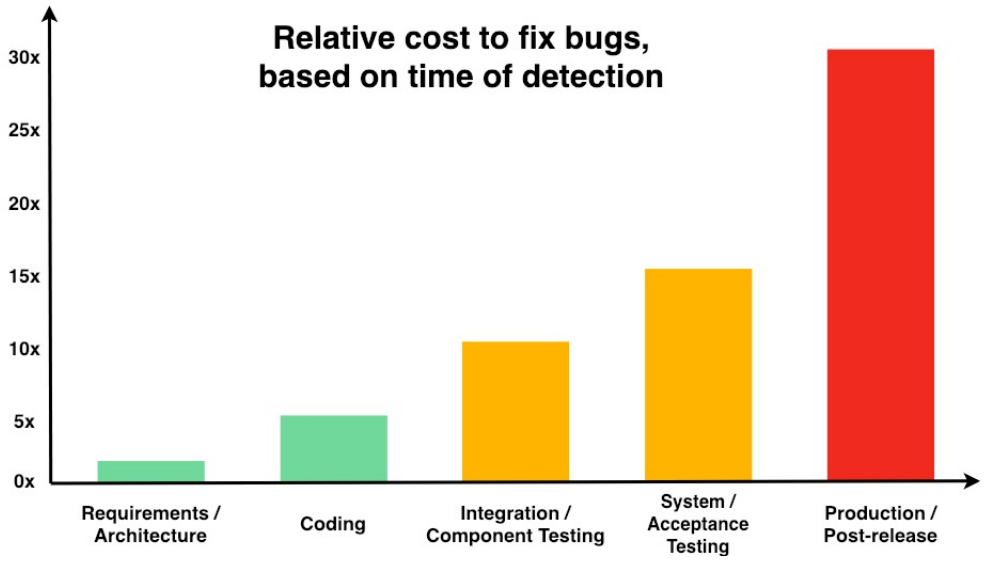


Abbildung 2.2: Kostenentwicklung eines Fehlers

Wie in der Abbildung 2.2 zu sehen ist, steigen die Kosten zur Behebung eines Fehlers aus den folgenden Gründen:

- Es ist einfacher Probleme oder Fehler im Code zu finden, wenn dieser beim Beheben noch frisch im Gedächtnis des Entwickler ist. Liegt die Entwicklung schon länger zurück, muss sich erst wieder in die Thematik eingelesen werden.
- Tritt ein Fehler in der Testphase auf, stellt das Reproduzieren, je nach Art des Fehlers, eine zeitaufwändige Aufgabe dar. Werden Fehler jedoch nicht in dieser Phase gefunden, treten diese in der Regel erst in der Produktionsphase wieder auf. Dies führt wiederum zu einem Anstieg der Kosten.
- Ist die Software bereits im Einsatz, ist es oft schwierig und riskant den Fehler zu finden und zu beheben. Neben der Vermeidung, dass Nutzer von den Problemen betroffen sind, kann die Sicherstellung der Verfügbarkeit des Dienstes geschäftsschädigend sein. Diese Auswirkungen summieren sich und können zu Kosten führen, welche eine frühzeitige Behebung des Fehlers um das 30-fache überschreiten können.[47]

2.2 Emulatoren und physische Geräte

Dieses Kapitel beschäftigt sich mit der Frage, ob Tests auf einem Emulator oder einem echten Gerät ausgeführt werden sollen, welche Wege es gibt, um eine Anwendung mit möglichst vielen Spezifikationen zu testen und was die wesentlichen Vor- und Nachteile eines Emulators sowie die eines physischen Geräts sind.

2.2.1 Emulatoren

Ein Emulator ist eine Software, welche die Eigenschaften von Hardware- oder Software-Systemen imitiert. Hierbei soll das Verhalten dem eines echten Systems entsprechen. Um die entwickelte Software direkt in der Entwicklungsumgebung zu testen, werden häufig Emulatoren des jeweiligen Systems verwendet.[3]

Das Verwenden eines Emulators bringt einige Vorteile mit sich. Der wohl größte ist, dass die meisten Emulatoren kostenlos zur Verfügung gestellt werden und sich an die

physischen Geräte anpassen lassen. Bei den in Android Studio mitgelieferten Emulatoren kann so zum Beispiel die Größe des Displays, Auflösung und Android-Version sowie die Größe des Arbeitsspeichers und des internen Speichers angepasst werden. Android Studio ist eine frei Integrierte Entwicklungsumgebung von Goolen und wird in der Entwicklung für Android-Softwareentwicklung verwendet. Als Extra wird in Android Studio das Analysetool „Android Profiler“ mitgeliefert, welches erlaubt die Auslastung der CPU und des Arbeitsspeichers sowie den Netzwerkverkehr und Stromverbrauch auszulesen.[17] Des Weiteren können auf einem Emulator sowohl manuelle als auch automatisierte Tests durchgeführt werden, die es ermöglichen den Großteil der üblichen Fehler abzufangen. Zudem muss ein Emulator nicht gewartet oder geladen werden. Jedoch kann dies auch zu einem Nachteil werden, da durch das stabile Umfeld keine natürlichen Akkuprobleme oder Netzwerkabbrüche auftreten. Andere Echtzeitdaten wie das GPS, Sensoren, Gesten oder die Berührungskraft können nicht nachgestellt werden. Natürlich wirkende Verhalten können zwar zum Teil festgelegt werden, entsprechen aber nicht einer echten Umgebung. Auch verschiedene Lichtbedingungen und deren Einfluss auf die Farbe und Kontrast des Displays kann nicht mit Hilfe eines Emulators nachgestellt werden. Äußere Einflüsse wie ein Anruf oder eine SMS kommen nicht ungeplant vor und Probleme mit dem Touchscreen können ebenfalls nicht emuliert werden. Emulatoren neigen zudem dazu, langsamer als die Originalgeräte zu sein, da die Taktung dem Rechner angepasst ist und diese nicht immer dem des echten Gerätes entspricht. Um möglichst wenig Probleme mit der Speicherkapazität zu haben, verfügen Emulatoren in der Regel über weitaus mehr Speicher als ein echtes Gerät. Besonders bei älteren Geräten sollte daher darauf geachtet werden, dass der Emulator in etwa denselben Speicher hat wie das Gerät, auf dem die Software ausgeführt werden soll.[28]

2.2.2 Physische Gerät

Der Preis und die Beschaffung eines physischen Gerätes sind wohl die größten Punkte, welche gegen eine ausschließliche Nutzung solcher sprechen. Durch die Kombination aus den elf Android Versionen und den unzähligen Displaygrößen sowie deren Auflösung, ist es unmöglich oder sehr kostspielig alle Spezifikationen zu besitzen. Jedoch gibt es keine bessere Möglichkeit die Interaktion zwischen Benutzer und der Anwendung zu prüfen, als diese auf einem echten Gerät zu installieren und anschließend zu testen. Alle realen Szenarien wie ein Netzwerkausfall, Hardwareprobleme oder auch verschiedene Lichtbedingungen können ungeplant auftreten und berücksichtigt werden. Mit dem Gerät oder

der Umgebung zusammenhängende Leistungsmängel werden so sichtbar. Zudem kann sich darauf verlassen werden, dass alle Ereignisse und Vorkommnisse denen eines echten Geräts entsprechen.

2.2.3 Geräte-Cloud

Um ein möglichst breites Feld an Spezifikationen der physischen Geräte/Emulatoren und Betriebssystemeen in einer eigenen Geräte-Cloud zu erhalten, ist eine größere Investition vonnöten. Es gibt einige Unternehmen, die diese Schwierigkeit erkannt haben und Lösungen anbieten. So bieten zum Beispiel AWS Device Farm, Firebase Test Lab oder Smartphone Test Farm ihre Dienste an. Hier können virtuelle oder reale Geräte gemietet, Tests parallel ausgeführt und deren Ergebnisse aufgezeichnet werden. Die Abrechnung erfolgt meist nach Betriebsstunden der Geräte. Werden nur wenige Testläufe pro Woche durchgeführt, kann es sich für kleinere Unternehmen durchaus lohnen solche Geräte zu mieten, da hier preiswert eine große Auswahl angeboten wird.[16] Eine eigene Geräte-Cloud senkt die laufenden Kosten, ist jedoch mit einer größeren Investition verbunden.

2.2.4 Diskussion

Nach Abwägung der Vor- und Nachteile zwischen Emulatoren und realen Geräten liegt die Schlussfolgerung nahe, dass die optimale Lösung eine Kombination aus beiden Systemen ist. Bei der Wahl des Systems werden typischerweise Emulatoren in der Entwicklungsphase eingesetzt und später mit realen Geräten ergänzt. Hierbei sollten unterschiedliche Modelle mit verschiedenen Android-Versionen verwendet werden, um ein realistisches Ergebnis zu erzielen. Um Kosten zu sparen, kann der Service einer Geräte-Cloud in Betracht gezogen werden. Ob diese Möglichkeit verwendet oder eine eigene Geräte-Cloud aufgebaut wird, hängt meist von der Zukunftsplanung des Unternehmens ab. Im Rahmen dieser Arbeit werden lokale Emulatoren für die automatisierten Test verwendet. Echte Geräte kommen bei der Finalisierung der Software zum Einsatz, um auch die reale Bedienbarkeit und das Verhalten zu überprüfen.

2.3 Stand der Technik

Dieses Kapitel behandelt Methoden und Strategien, welche empfohlen oder vorgegeben werden, um Softwareprojekte erfolgreich zu testen.

2.3.1 Teststrategien

"Das Testen von Programmen kann die Existenz von Fehlern zeigen, aber niemals deren Nichtvorhandensein!" Edsger W. Dijkstra. [10]

Da es in der Praxis aufgrund des Umfangs nicht realistisch ist, einen vollständigen Test zu schreiben, ist es unumgänglich, eine Teststrategie zu wählen. Ein vollständiger Test prüft das Gesamtsystem unter allen Vorbedingungen und Eingabemöglichkeiten. Die Teststrategie sollte in der Testplanung anhand einer Risikoabschätzung festgelegt werden. Dabei ist zu beachten, wie kritisch das Auftreten eines Fehlers in einem Systemteil einzuschätzen ist und wie intensiv ein System getestet werden kann oder muss. [41]

Da die Software, welche im Rahmen dieser Arbeit getestet wird, bereits zu einem großen Teil besteht und unabhängig der Teststrategie und Praxis entwickelt wird, soll ein Ausblick gegeben werden, welche Philosophie, Praxis, Herangehensweise oder Idee hinter den jeweiligen Strategien steht. In zukünftigen Projekten kann diese von Beginn an angewendet werden.

2.3.2 Testzyklen

Die Einordnung der Teststufen oder Testzyklen folgt in mehreren Abwandlungen dem V-Modell, welches 1979 von Barry Boehm vorgeschlagen wurde und auf dem Wasserfallmodell basiert. [35]

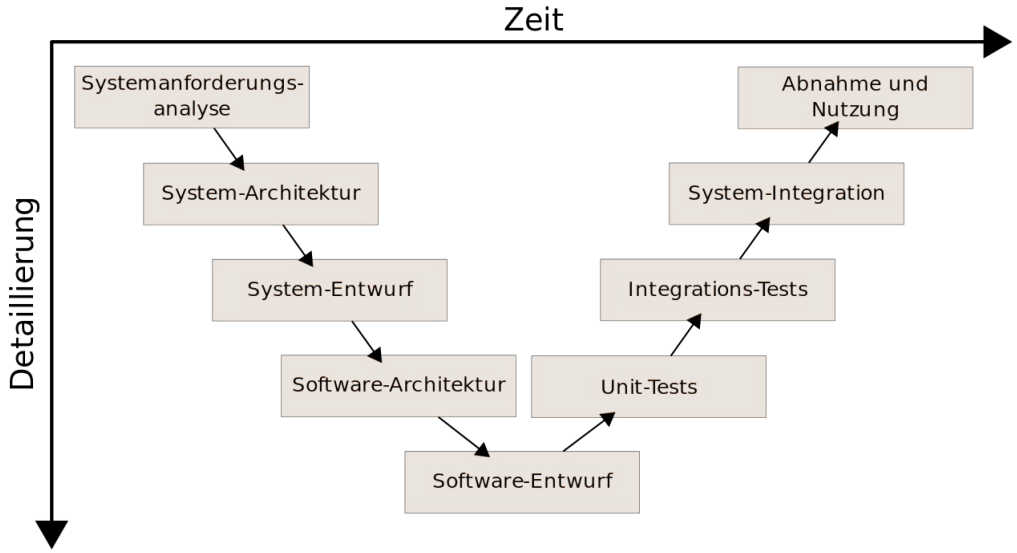


Abbildung 2.3: V-Modell

Die Wasserfallmethode ist eine Projektstrategie innerhalb der Informatik, bei der die einzelnen Prozessschritte nacheinander abgearbeitet, überprüft, von dem Kunden abgenommen und freigegeben werden. Im Vergleich dazu wird bei einem iterativen Vorgehen, was ebenfalls eine Projektstrategie ist, die Software stetig optimiert und neuen Anforderungen angepasst. In beiden Strategien können Testzyklen mit Hilfe des V-Modells geplant und durchgeführt werden. Das V-Modell ist ein Vorgehensmodell, dass für die Softwareentwicklung konzipiert wurde und die Organisation und Qualitätssicherung in einem Softwareentwicklungsprozess vorgibt. Auf der rechten Seite des V-Modells befinden sich die jeweiligen Teststufen. Auf der gegenüberliegenden Seite der dazugehörige Systementwurf. Dies zeigt, dass die jeweiligen Testfälle und Ziele auf dem dazugehörigen Entwicklungsergebnis basieren. Bei einem iterativen Vorgehen ist dies jedoch nur möglich, wenn in jeder Iterationsstufe die Änderungen in dem gesamten System und somit auch den Tests und deren Spezifikationen angepasst und umgesetzt werden. Dies ist bei einem Projekt nicht notwendig, welches mit Hilfe der Wasserfallmethode umgesetzt wird, da der Entwicklungsprozess einmal von vorne nach hinten durchgeführt wird. Bei der iterativen Methode ist es vorgesehen, dass nach jedem Entwicklungszyklus eine lauffähige Software ausgerollt werden kann. Das führt dazu, dass die Testaktivität in kleineren Einheiten aber dafür in größerer Anzahl geplant und durchgeführt werden. Hierbei kommen alle Testarten bis hin zu funktionalen und nicht-funktionalen Systemtest zum Einsatz. Nach einem Sprint werden zudem alle Tests aus den vorherigen Sprints ausgeführt. So

kann sichergestellt werden, dass es zu keinen Konflikten zwischen dem alten und neuen Code kommt.

Um einen Leitfaden zu erhalten, in welchem Umfang und Art die Software getestet werden sollte, gibt es einige Modelle. Eines davon ist die Testpyramide, durch die eine definierte Reihenfolge entsteht, in welcher die Tests entwickelt und durchgeführt werden sollen, um ein möglichst ausgewogenes und umfassendes Portfolio an Tests zu erhalten: Modultests (Kapitel 3), Integrationstests (Kapitel 4), Tests der Benutzeroberfläche (Kapitel 5). [15]

2.3.3 Testpyramide

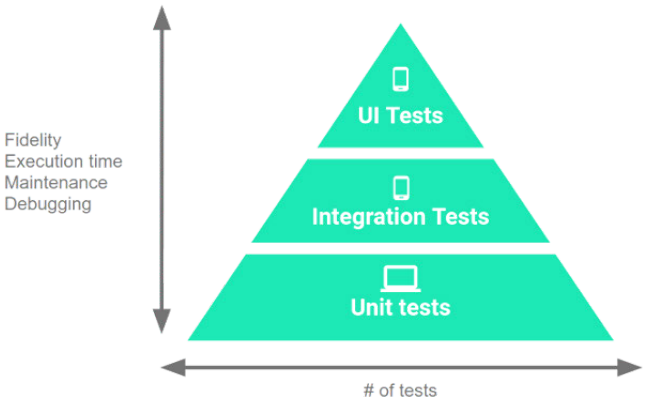


Abbildung 2.4: Testpyramide

Die Testpyramide beschreibt wie eine Anwendung in drei unterschiedlich große Testkategorien aufgeteilt werden kann. Die jeweilige Größe der Testkategorie besagt in welchem Umfang sie erstellt und in welcher Häufigkeit diese ausgeführt werden sollen. Im Vergleich zu der Größe der Kategorie steigt die Ausführungszeit und der Aufwand für die Fehlersuche und Wartung umgekehrt proportional. [1] Die jeweiligen Ebenen werden im Verlauf dieser Arbeit genauer betrachtet.

In der untersten und somit größten Ebene befinden sich die Modultests, welche für jedes Modul der Anwendung das Verhalten validieren. In der mittleren Ebene befinden sich die Integration-Tests.

Mit diesen Tests wird die Interaktion in einem Modul zwischen verwandten Modulen oder die Interaktion zwischen den Ebenen des Stacks überprüft. Bei Stacks handelt es sich um eine Reihe von aufeinander aufbauenden Softwarekomponenten, die gemeinsam eine Plattform bilden. An der Spitze der Pyramide stehen die UI-Tests oder auch

Benutzeroberflächen-Tests. Diese prüfen die Navigation durch die Anwendung, welche mehrere Module umfassen können. [25]

2.3.4 Testgetriebene Entwicklung

Eine weitere Praxis um Software zu testen ist, die testgetriebene Entwicklung. Diese wurde in den späten 90ern von Kent Beck geprägt und ist ein Teil der Entwicklungsmethode „Extreme Programming“ und wird häufig in der agilen Softwareentwicklung angewandt. Trotz des hohen Alters von mehreren Jahrzehnten wird der Praxis noch immer eine große Relevanz zugeschrieben. Um ein genaueres Bild der testgetriebenen Entwicklung zu erhalten, lässt sich diese in die folgenden drei Aspekte unterteilen: die Tests, das Getriebene und die Entwicklung.

Die Tests beinhalten das Schreiben von automatisierten Tests für die einzelnen Module eines Programmes. Bei den Tests handelt es sich um Modultests, welche in der Praxis von den Entwicklern in kleinen und schnellen Iterationen geschrieben werden und das Testen eines Moduls mit nur einem Mausklick ermöglichen. Dies führt zu einer hohen Ausführungsrate der Tests, wodurch Fehler schneller gefunden und behoben werden können. Auf eine genauere Definition, Vorgehensweise und Anwendung der Modultests wird in Kapitel 3 eingegangen.

Der angetriebene Aspekt beinhaltet, dass der Entwickler dazu angewiesen ist, Code nur dann zu schreiben, wenn ein Test fehlgeschlagen ist. Hierfür wird vorausgesetzt, dass für jedes bisschen an Funktionalität im Code zunächst ein Test geschrieben wird, welcher spezifiziert und validiert, was der Code erfüllen soll. Sind die Tests geschrieben, wird genau so viel Code geschrieben, bis diese erfüllt sind. Um das Testen zu einem Vorgang der Analyse und des Designs zu machen, wird der Schritt der Restrukturierung durchgeführt, bei welchem es sich um eine Technik zur Neustrukturierung des bereits bestehendes Quellcodes handelt. Hierbei wird die interne Struktur angepasst, ohne das Verhalten nach außen zu verändern. Mit diesem Schritt kann Quellcode, welcher die Tests zwar besteht, jedoch unnötig komplex oder unflexibel ist, zu einem sauberen Code umgewandelt werden. Dieser besteht die Tests noch immer, ist aber qualitativ hochwertiger oder funktionaler, wodurch die Punkte Lesbarkeit, Verständlichkeit, Wartbarkeit und Erweiterbarkeit verbessert werden. Der häufigste Fehler, welcher bei der testgetriebenen Entwicklung begangen wird und somit zu einem Scheitern der Methodik führt, ist, dass der Schritt der Restrukturierung vernachlässigt oder ausgelassen wird.

Bei der Entwicklung ist zu beachten, dass die testgetriebene Entwicklung bei der Konstruktion von Software helfen soll und an sich keine Entwicklungsmethodik oder Prozessmodell ist. Es handelt sich eher um eine Art und Weise oder Praxis, bei welcher andere Praktiken in einer bestimmten Reihenfolge und Häufigkeit innerhalb eines Prozessmodells verwendet werden und kann als Mikroprozess im Kontext vieler verschiedener Prozessmodelle angewendet werden.[30]

Die zwei großen Vorteile, welche man durch die Nutzung von der testgetriebenen Entwicklung erhält, sind, dass durch das Nachdenken über die Tests zuerst über die Schnittstellen zum Code nachgedacht werden muss. Dieser Fokus auf die Schnittstelle und die Art und Weise, wie eine Klasse verwendet wird, hilft dabei die Schnittstelle vor der Implementierung zu trennen. Der zweite Vorteil ist, dass Funktionen oder Module nur auf der Basis von Tests geschrieben werden und so ein sich selbsttestender Code erzeugt wird, auf welchen im Folgenden eingegangen wird. [12][30]

2.3.5 Selbsttestender Code

Selbsttestender Code beschreibt die Praxis des Schreibens von umfassenden automatisierten Tests in Verbindung mit der funktionalen Software, wodurch ein Fehlerdetektor entsteht, welcher wiederum in der Lage ist, Fehler innerhalb des Systems aufzuspüren. So kann man mit einem einzelnen Befehl den Vorgang des Testens starten. Sollte dieser erfolgreich abgeschlossen werden, kann man davon ausgehen, dass die meisten der versteckten Fehler in dem Code gefunden wurden. Aufgrund der schnellen Ausführung der Tests werden diese im besten Fall mehrmals am Tag ausgeführt, damit Fehler zeitnah entdeckt und behoben werden können.

Von selbsttestendem Code wird häufig im Zusammenhang mit testgetriebener Entwicklung gesprochen, wobei dies getrennt betrachtet werden sollte. So ist es bei dieser Praxis nicht relevant, ob die Tests vor oder nach dem Entwickeln des Moduls oder der Software geschrieben werden. Wichtig ist nur, dass sie geschrieben werden. Zudem ist ein selbsttestender Code ein Schlüsselelement der kontinuierlichen Integration (Kapitel 6), da sichergestellt werden kann, dass auftretende Fehler im Prozess der kontinuierlichen Integration zum Abbruch dieser führen und das Entwicklerteam über den Fehler informiert wird.

Um selbsttestenden Code zu erhalten sollte jedes Objekt in der Lage sein, sich selbst zu testen. Dies hat zur Folge, dass nach jeder Freigabe eine lauffähige und ausführlich getes-

tete Software entsteht. Dies kann wiederum dazu führen, dass Vertrauen gewonnen wird, da nach dem Bestehen der Tests davon ausgegangen werden kann, dass die Änderungen an einem Objekt oder dem System keine negativen Auswirkungen haben.

Kommt es bei der Ausführung der Tests zu einem Fehler, wird zunächst ein Test geschrieben, welcher den Fehler zuverlässig reproduziert. Um die Ursache des Fehlers einzugrenzen, werden zunächst eine Reihe an Tests geschrieben. An letzter Stelle sollte ein Modultest stehen, der den Fehler auslöst. Erst dann wird dieser behoben. Der gefundene Fehler dient nicht nur zur Behebung desjenigen, sondern wird auch genutzt, um das Testschema stetig zu optimieren und zu erweitern, um ähnliche Fehler aufzudecken. [14]

2.4 Mockups

In dem folgenden Kapitel wird der Begriff des Mockups erklärt und die Verwendung sowie die Implementierung der Bibliothek "MockK" aufgezeigt.

2.4.1 Grundlagen

Ein Mockup kann wie folgt beschrieben werden: *„Ein Modell von etwas, das zeigt, wie etwas aussehen oder funktionieren wird, wenn das Echte noch nicht verfügbar ist.“*. [9]

Je nach Art des Tests sollten die zu testenden Komponenten, Objekte, Klassen oder Funktionen alleinstehend sein. Das bedeutet, dass der jeweilige Test durchgeführt wird, ohne dabei von anderen Komponenten abhängig zu sein. Dies kann zum Beispiel daran liegen, dass es von der Teststrategie vorgegeben wird, dass die Komponente zu dem Zeitpunkt des Tests nicht verfügbar ist, die Ergebnisse nicht relevant sind oder einen unerwünschten Nebeneffekt hervorrufen. Ebenfalls können durch Mockups Komponenten in den Test einbezogen werden, obwohl diese noch nicht entwickelt wurden.

So muss in einer Klasse, in welcher Informationen eines Benutzers geprüft werden, die Methode `age()` auf einige Funktionen der Nutzer- oder Kalender-Klasse zugreifen. Sollten die Tests alleinstehend sein, kann die echte Nutzer- oder Kalender-Klasse nicht verwendet werden, da ein Fehler in den Klassen dazu führen würde, dass die Tests der ursprünglichen Klasse fehlschlagen. In diesem Fall würde es sich um einen gesellschaftsfähigen Test handeln, welcher verwendet werden kann, sollte die Kommunikation mit den

verwendeten Komponenten nicht zu umständlich sein. Hierzu muss jedoch die Annahme getroffen werden, dass alle anderen Komponenten funktionieren, was die Fehlersuche erschwert.

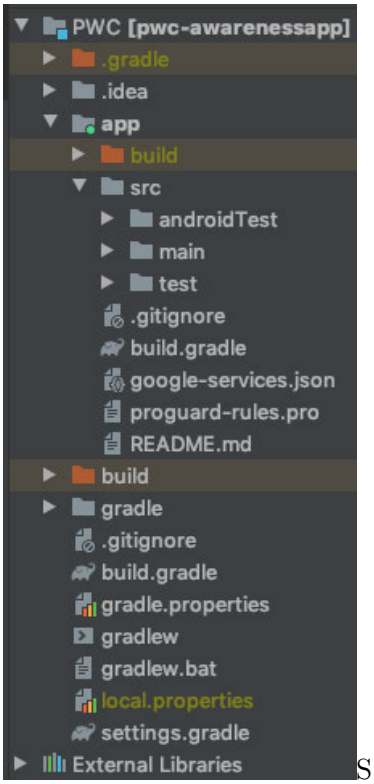
Um einen alleinstehenden Test zu erhalten, werden Test-Double verwendet. Bei Test-Doubles handelt es sich um einen von Gerard Meszaros geprägten Namen und beschreibt einen Oberbegriff für alle Fälle, in denen für Testzwecke ein Objekt ersetzt wird. [13]

Test-Doubles

- Dummy-Objekte werden weitergeschoben, aber nie wirklich verwendet. Sie werden meist zum Füllen von Parameterlisten verwendet. [13]
- Fake-Objekte sind Objekte, welche eine tatsächlich funktionierende Implementierungen beinhalten, jedoch nicht den vollen Umfang der tatsächlichen Funktion haben oder nicht mit dem Produktionscode übereinstimmen. Ein Beispiel hierfür ist eine Fake-Implementierung für einen Zugriff auf eine Datenbank, wodurch der Test nicht auf das Ergebnis der Datenbank warten muss, sondern vorgefertigte Werte verwendet. So kann ein Test durchgeführt werden, ohne eine Datenbank starten zu müssen. Zudem kann die Dauer der Tests optimiert werden, da zeitaufwändige Anfragen nicht real durchgeführt werden müssen. [36]
- Stubs liefern vordefinierte Antworten auf Aufrufe, welche während des Tests getätigt werden, können jedoch nicht reagieren, wenn die Anforderung außerhalb des für den Test programmierten Bereich liegen.
- Spies sind Stubs, die abhängig von der Art des Aufrufes Informationen aufzeichnen. Die aufgezeichneten Informationen können im Nachhinein ausgewertet werden. [13]
- Mocks sind mit Erwartungen vorprogrammierte Objekte, welche eine Spezifikation der Aufrufe bilden, die sie erhalten sollen. Sie können einen Fehler werfen, sollten sie einen Aufruf erhalten, welchen sie nicht erwarten. Zudem wird während der Überprüfung verifiziert, ob alle erwarteten Aufrufe getätigt wurden. [13]

2.4.2 MockK

MockK ist eine öffentliche Bibliothek, welche zu einem großen Teil von Oleksiyp entwickelt wurde. [46] MockK basiert auf dem Prinzip von Mockito, was wiederum eine Bibliothek zum Erstellen von Test-Doubles ist. MockK basiert im Vergleich zu Mockito auf Kotlin. Da das zu testende Projekt ebenfalls in Kotlin geschrieben wird, sind die Tests-Doubles in dieser Arbeit mit Hilfe von MockK umgesetzt. Die meisten JVM Mock-Bibliotheken haben ein Problem finale Klassen zu Stubben oder Mocken. Dies stellt ein Problem dar, da in Kotlin alle Klassen und Methoden final sind. Um diese dennoch testen zu können, ohne sie umschreiben zu müssen, ist es sinnvoll, eine Bibliothek zu nutzen, welche die Spracheigenschaften von Kotlin unterstützt.



S

Abbildung 2.5: Android Struktur

Um MockK in Android verwenden zu können, wird die Abhängigkeit zu der MockK-Bibliothek in `PWC/app/build.gradle` hinterlegt.[24]

Für die Verwendung in den Modultests, welche sich in einem Android Projekt in dem automatisch erzeugten Verzeichnis `PWC/app/src/test` befinden, wird bei der Ab-

hängigkeit „testImplementation“ angegeben. Somit können nur Modultests darauf zugreifen. Mit der Angabe „androidTestImplementation“ ist die Bibliothek für die Integrationstests und UI-Tests verfügbar, die sich in diesem Projekt unter dem Verzeichnis `PWC/app/src/androidTest` befinden. Um Coroutines zu mocken, wird der Support-Bibliothek zusätzlich die Abhängigkeit `org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.9` hinzugefügt.

```
dependencies {
    // for Instrumented-Tests
    testImplementation "io.mockk:mockk-android:1.10.0"
    // for Unit-Tests
    testImplementation "io.mockk:mockk:1.10.0"

}
```

Quellcode 2.1: MockK Dependency

Beispiel

Um ein Mock-Objekt der Klasse MockKHStore zu erzeugen, wird die Methode `mockk<MockKHStore>()` verwendet und der Rückgabewert für den Aufruf der Funktion mit dem Parameter ”Param” mit Hilfe von `every {store.getDataFromDB(”Param”} returns ”Output”` definiert. Anschließend wird die Funktion aufgerufen und in der Variablen ”result” gespeichert. Mit Hilfe von `verify { store.getDataFromDB(”Param”)}` wird anschließend überprüft, ob der Mock wie erwartet aufgerufen wurde. Dann wird mit `assertEquals(”Output”, result)` überprüft, ob der zurückgegebene Wert dem erwarteten Wert entspricht. Ist dies der Fall, ist der Test bestanden.

```
class MockKHStore() {
    fun getDataFromDB(path: String): String {
        return "some String from DB"
    }

    fun getOtherData(path: String): String {
        return "other String"
    }
```

```
}

@Test
fun testFun() {
    val store = mockk<MockKHStore>()
    every { store.getDataFromDB("Param") } returns "Output"

    val result = store.getDataFromDB("Param")
    verify { store.getDataFromDB("Param") }
    assertEquals("Output", result)
}
```

Quellcode 2.2: MockK Beispiel

Notation

Die Notation `@MockK` gibt ein Objekt an, welches definierte Mocks enthält, die jedoch noch nicht zugewiesen sind. MockK erfordert zudem, dass auf einem Objekt, welches eine Variable mit Annotation deklariert, `MockKAnnotation.init(...)` aufgerufen wird. Dies wird in `@Before`, `@BeforeEach` oder im `init` einer Variablen ausgeführt.

```
@MockK
lateinit var type1: MockKHStore
```

Quellcode 2.3: MockK Notation

```
@Before
fun setUp() {
    MockKAnnotations.init(this)
}
```

Quellcode 2.4: MockK Init Notation

Auf die Notationen `@RelaxedMockK`, `@MockK(relaxUnitFun = true)`, `@SpyK` und `@InjectMockKs` wird in den folgenden Punkten eingegangen.

Entspannter Mock

Ein entspannter Mock ist ein Mock, welcher einen einfachen Wert für alle Funktionen zurückgibt. Dies führt dazu, dass einzelne Verhalten festgelegt werden können, jedoch nicht für alle Funktionen vonnöten sind. Sollte es sich um einen Referenztyp handeln, wird ein verketteter Mock zurückgegeben. Bei generischen Rückgabetypen sollte jedoch der Stub manuell erzeugt werden, da sonst eine Class-Cast-Exception geworfen wird. Um einen Mock mit dieser Eigenschaft zu erhalten, kann eine der folgenden Notationen verwendet werden:

```
@Test
fun relaxedTest() {
    val hStore = mockk<MockKHStore>(relaxed = true)
}

@MockK(relaxed = true)
lateinit var relaxedStore: MockKHStore

init {
    MockKAnnotations.init(this)
}

@MockK
lateinit var relaxedStore2: MockKHStore

init {
    MockKAnnotations.init(this, relaxed = true)
}
```

Quellcode 2.5: Entspannter Mock

Falls eine Funktion in einem Modultest entspannt sein soll, wird eine der Notation verwendet.

```
@Test
fun unitMock() {
    val store = mockk<MockKHStore>(relaxUnitFun = true)
}
```

```
@RelaxedMockK
lateinit var unitStore1: MockKHStore

@MockK(relaxUnitFun = true)
lateinit var unitStore2: MockKHStore

init {
    MockKAnnotations.init(this)
}

@MockK
lateinit var unitStore3: MockKHStore

init {
    MockKAnnotations.init(this, relaxUnitFun = true)
}
```

Quellcode 2.6: Entspannter Unit Mock

Spies

Spies erlauben Mocks und reale Objekte zu kombinieren, wobei das Spy-Objekt eine Kopie des übergebenen Objektes ist und mittels `@SpyK lateinit var store: MockKHStore` oder `spyk<MockKHstore>()` erzeugt werden kann. Mit `every { store.getDataFromDB(any( )) }` `retuns "Output"` wird festgelegt, dass die Funktion `getDataFromDB()` immer den String `"Output"` zurückgibt. Alle weiteren Funktionen werden nicht gemockt und geben daher den tatsächlichen Wert zurück. Um die Ergebnisse zu überprüfen, wird mit `assertEquals (first, "Output")` der gemockte Wert und mit `assertEquals (second, "Output")` der Wert der nicht gemockten Funktion überprüft, welcher dem realen Wert entspricht.

```
@SpyK
lateinit var store: MockKHStore
```

```
@Test
fun spyTest() {
    val store = spyk<MockKHStore>()
    every { store.getDataFromDB(any()) } returns "Output"

    val first = store.getDataFromDB("Any Param")
    assertEquals(first, "Output")

    val second = store.getOtherData("Any Param")
    assertEquals("other String", second)
}
```

Quellcode 2.7: Spies Beispiel

Mock-Objekt

Das Mock-Object von MockStore wird mit Hilfe von `mockkObject(MockStore)` erzeugt. Im nächsten Schritt wird mit `assertEquals(true, realState)` überprüft, ob in dem gemockten Objekt der reale Wert in der Variablen `connected` steht. Anschließend wird `connected` der Wert `false` zugewiesen und ebenfalls überprüft. Entspricht der gemockte Wert dem erwarteten, ist der Test bestanden.

```
object MockStore {
    var connected = true
}

@Test
fun objectTest() {
    mockkObject(MockStore)
    val realState = MockStore.connected
    assertEquals(true, realState)

    every { Store.connected } returns false
    val mockedState = MockStore.connected
    assertEquals(false, mockedState)
}
```

Quellcode 2.8: Mock-Objekt Beispiel

Um das Objekt in den Ursprungszustand zu versetzen, gibt es zwei Funktionen. Mit `unmockkAll()` werden alle gemockten Objekte zurückgesetzt.
`unmockkObject(MockStore)` versetzt das angegebene Objekt in den Ursprungszustand. Beide Funktionen können direkt in der Funktion des Tests durchgeführt werden oder in einer mit `@After` deklarierten Funktion, welche nach jedem Test ausgeführt wird.

```
@After
fun cleanUp(){
    unmockkAll()
    unmockkObject(MockStore)
}
```

Quellcode 2.9: Unmock-Objekt Beispiel

Klassen Mock

Um Klassen zu mocken, wird die Notation `mockkClass(Test::class)` verwendet. Auch hier wird mit Hilfe von `every {classMock.getDataFromDB("Param")} returns "Output"` bestimmt, welchen Rückgabewert die Funktion in der Klasse hat. Anschließend wird mit `assertEquals("Output", result)` überprüft, ob das Mocken der Funktion in der Klasse wie erwartet funktioniert. Ist dies der Fall, ist der Test bestanden. $

```
fun classTest(){
    val classMock = mockkClass(MockKHStore::class)
    every { classMock.getDataFromDB("Param") }
            returns "Output"
    val result = classMock.getDataFromDB("Param")
    assertEquals("Output", result)
}
```

Quellcode 2.10: Klassen Mock Beispiel

Hierarchischer Mock

Eine weitere nützliche Funktion von MockK ist die Möglichkeit, hierarchische Objekte zu erzeugen. Hierbei enthält das Objekt `User` das Feld `City` und kann wie in 2.11

zu sehen innerhalb des Mocks des Benutzers erzeugt werden. Anschließenden wird mit `assertEquals ( "Simon" , name)` überprüft, ob der Name des Users und mit `assertEquals ( "Dort" , cityName)` der Name der Stadt richtig gemockt wurde.

```
class User{
    lateinit var name: String
    lateinit var city: City
}

class City {
    lateinit var name: String
}

@Test
fun hierarchicalMockTest() {
    val user = mockk<User> {
        every { name } returns "Simon"
        every { city } returns mockk {
            every { name } returns "Dort"
        }
    }

    val name = user.name
    val cityName = user.city.name

    assertEquals("Simon", name)
    assertEquals("Dort", cityName)
}
```

Quellcode 2.11: Hierarchischer Mock Beispiel

Erfassen von Parametern

Um die übergebenen Parameter an eine Methode zu erfassen, kann für einzelne Werte ein Slot, für mehrere Werte eine MutableList verwendet werde.

Um einen übergebenen Parameter zu prüfen, wird zunächst der Variable `slot = slot<String>()` zugewiesen. Anschließend wird im Aufruf der Funktion `getDataFromDB()` der übergebene Parameter wie folgt gespeichert: `every{store.getDataFromDB(capture(slot)} returns "Output"`. Wird anschließend die Funktion aufgerufen, kann der übergebene Parameter mit Hilfe von `slot.captured` überprüft werden.

```
@Test
fun capturingSlotTest() {
    val store = mockk<MockKHStore>()
    val slot = slot<String>()
    every { store.getDataFromDB(capture(slot)) }
            returns "Output"
    store.getDataFromDB("Value")
    assertEquals("Value", slot.captured)
}
```

Quellcode 2.12: CapturingSlot Beispiel

Wird die Funktion mehrmals aufgerufen, werden die Werte in einer MutableList gespeichert. Anstelle von `slot = slot<String>()` wird der Variable eine Liste aus Strings zugewiesen `list = mutableListOf<String>()`. Nachdem die Funktion mehrmals aufgerufen wurde, kann in der Liste der jeweilige Parameter überprüft werden. Die Position in der Liste entspricht hierbei der Reihenfolge der Aufrufe.

```
@Test
fun mutableListParamsMock() {
    val store = mockk<MockKHStore>()
    val list = mutableListOf<String>()
    every { store.getDataFromDB(capture(list)) }
            returns "Output"
    store.getDataFromDB("Value␣1")
    store.getDataFromDB("Value␣2")
    assertEquals(2, list.size)
    assertEquals("Value␣1", list[0])
    assertEquals("Value␣2", list[1])
}
```

Quellcode 2.13: MutableList Beispiel

3 Modultests

Dieses Kapitel behandelt die Modultests, was deren Vorteile sind und wie sie angewandt werden sollten. Anschließend wird gezeigt, wie diese in dem Projekt umgesetzt werden und welche Problematiken es gibt.

3.1 Grundlagen

Der Umfang der Modultests sollte in etwa 70 Prozent der gesamten Tests ausmachen und wird in der Praxis meist von den jeweiligen Entwicklern mit ihren regulären Werkzeugen geschrieben.[25] In der testgetriebenen Entwicklung werden diese vor der Programmierung des Moduls erstellt. Durch einen kleinen Umfang und der daraus resultierenden kurzen Laufzeit, können und sollen Modultests häufig durchgeführt werden um Fehler frühzeitig zu erkennen. Sollte es bei der Übergabe an die Versionsverwaltung zu einem Fehler kommen, liegt die Fehlerquelle meist an den Änderungen der letzten Stunden oder des Tages. Im Vergleich dazu kann die Änderung, welche einen Fehler in den Integrationstests oder den UI-Tests hervorruft, bereits Tage oder Wochen zurückliegen. Jedoch kann nicht nur durch das häufige ausführen Zeit gespart werden. Meist kostet es mehr Zeit, ein Modul mit allen Eingaben manuell zu testen als einen Modultest zu schreiben. Da die Kosten, um einen Fehler zu beheben, mit jedem Entwicklungsschritt ansteigen (Abschnitt 2.1), kann mit ausführlichen Modultests Zeit und somit Geld gespart werden. Spätestens wenn der erstellte Code an das System zur Versionsverwaltung übergeben wird, sollte ein gesamtheitlicher Modultest durchgeführt werden, in welchem sich alle einzelnen Modultests befinden. Dies sollte in einem zeitlichen Rahmen von unter einer Sekunde bis hin zu zehn Minuten befinden. Dies führt dazu, dass man einen gesamtheitlichen Modultest auch während kurzen Pausen und mehrmals am Tag laufen lassen kann. [11] Sind alle Modultests bestanden, kann dies das Vertrauen in den entwickelten Code erhöhen, da Fehler in dem Modul aufgezeigt werden. Um eine Umgebung für die

Modultests zu schaffen ist es in der Umgebung von Android empfehlenswert, Frameworks wie Robolectric zu verwenden.

3.2 Robolectric

Robolectric kann verwendet werden, um Tests innerhalb einer JVM auszuführen. Im gegensatz zu Tests, welche man auf einem herkömmlichen Emulator oder physischen Gerät ausführt, laufen Robolectric-Tests innerhalb einer Sandbox. Hierdurch wird es ermöglicht, die Umgebung von Android auf die gewünschten Bedingungen zu konfigurieren. So kann man die Lebenszyklen von Komponenten simulieren und Ereignisschleifen ausführen. Zudem kann auf alle Ressourcen, welche sich in `pwc/app/src/main/res` befinden, zugegriffen werden. Durch Test-Double, welche von der Gemeinschaft gepflegt und Shadows genannt werden, kann der vom Framework abhängige Code getestet werden, ohne Mocks verwenden zu müssen. Dies führt meist dazu, dass keine rückwirkende Implementierung in dem Anwendungscode vonnöten ist. [25] Es besteht jedoch weiterhin die Möglichkeit, Mocking-Frameworks wie MockK zu verwenden.

Durch die Verwendung von Robolectric kann die Zeit, welche benötigt wird, um den Test auszuführen, von mehreren Minuten auf wenige Sekunden gesenkt werden. Dies kommt zustande, da auf dem Rechner und in der Umgebung der kontinuierlichen Integration (Kapitel 6) kein Emulator gestartet werden muss und die Schritte des Dexing, Paketieren und der Installation entfallen.[27] Dexing beschreibt hierbei den Vorgang der Konvertierung des Bytecodes in das native Format von Android.[26]

Durch die Abhängigkeit `testImplementation 'org.robolectric:robolectric:4.4'` in `build.gradle` kann Robolectric verwenden werden. Um auf die Ressourcen des Projektes, wie die hinterlegten Strings, Farben und Dimensionen, zugreifen zu können, wird unter `android{testOptions{unitTests {}}}` `includeAndroidResources = true` hinzugefügt. Ist dies hinterlegt, kann eine Testklasse mit der Notation `@RunWith(RobolectricTestRunner.class)` versehen werden. [27] Nachdem die Vorbereitung abgeschlossen ist, können Modultests implementiert werden.

3.3 Lokale Modultests

Lokale Modultests werden in dem von Android Studio selbst erzeugten Verzeichnis `pwc/app/src/test/java/x` angelegt. Zudem wird die Abhängigkeit für das JUnit4 - Framework unter dem Punkt `dependencies` in der `build.gradle` Datei hinterlegt. [19] Hierbei handelt es sich um das am häufigsten genutzte Framework für Modultests in Java. JUnit-Tests liefern als Ergebnis, ob der Test bestanden ist oder nicht. Sollte er nicht bestanden sein, kann dies daran liegen, dass ein falsches Ergebnis vorliegt oder ein Fehler geworfen wird. In beiden Fällen wird eine Exception geworfen. [33]

3.3.1 Validierung des Passworts

Im Folgenden wird die Funktion zur Validierung des Passworts getestet. Diese befindet sich in der Klasse `LoginViewModel` und ist ein Companion Objekt. Ein Companion Objekt wird verwendet, um Teile des Objektes über den Namen der Klasse erreichbar zu machen. [31] So kann die Funktion `isValidPassword(password: String?): Boolean` mit `LoginViewModel.isValidPassword(string)` aufgerufen werden.

```
companion object {
    fun isValidPassword(password: String?): Boolean {
        return Pattern.compile(
            regex: "^(?=.*[0-9])(?=.*[A-Z])(?=.*[!§()?`´*@#$%^&+=])(?=\\S+$).{8,}")
            .matcher( input: password ?: "")
            .matches()
    }
}
```

Abbildung 3.1: Funktion zur Validierung des Passwortes

Die Klasse wird zunächst mit der Notation `@RunWith(RobolectricTestRunner::class)` versehen, damit die Tests in der Umgebung von Robolectric gestartet werden können. Mithilfe von `@Config(sdk = [Build.VERSION_CODES.O_MR1])` kann festgelegt werden, mit welcher Version von Android der Test ausgeführt werden soll. Dies kann dabei helfen, neue Funktionen oder Features zu testen, welche erst ab einer vorgegebenen Version verfügbar sind.

```
@RunWith(RobolectricTestRunner::class)
@Config(sdk = [Build.VERSION_CODES.O_MR1])
class PasswordValidTest {

    @Test
    fun `password validation - junit`() {
        assertTrue(isValidPassword("T33*sasa"))
        assertTrue(isValidPassword("ls0D§dsA"))
        assertFalse(isValidPassword(""))
        assertFalse(isValidPassword("*ASsla2"))
        assertFalse(isValidPassword("*las9320sal"))
        assertFalse(isValidPassword("Adj93io123"))
        assertFalse(isValidPassword("*asdAsA§"))
        assertFalse(isValidPassword(null))
        assertTrue(isValidPassword("*WE4*A$§"))
    }

    @Test
    fun `password validation - truth`() {
        assertThat(isValidPassword("T33*sasa")).isTrue()
        assertThat(isValidPassword("ls0D§dsA")).isTrue()
        assertThat(isValidPassword("")).isFalse()
        assertThat(isValidPassword("*ASsla2")).isFalse()
        assertThat(isValidPassword("*las9320sal")).isFalse()
        assertThat(isValidPassword("Adj93io123")).isFalse()
        assertThat(isValidPassword("*asdAsA§")).isFalse()
        assertThat(isValidPassword(null)).isFalse()
        assertThat(isValidPassword("*WE4*A$§")).isTrue()
    }
}
```

Quellcode 3.1: Testklasse PasswordValidTest

Die Testmethode beginnt mit der Annotation `@Test` und enthält den Code zum Ausführen und überprüfen einer einzelnen Funktion. Die Vorgaben an das Passwort sind: es muss mindestens acht Zeichen lang sein und eine Zahl, einen Großbuchstaben und ein Sonderzeichen enthalten. Daraus ergeben sich die zu testenden Fälle, welche Fehlerhaft sind und somit `false` als Rückgabewert liefern sollten. Die zu übergebende Zeichenkette ist hierbei: leer, kürzer als acht Zeichen oder `null`, beinhaltet keinen Großbuchstaben, kein Sonderzeichen oder keine Zahl. Für den positiven Test werden zwei valide Zeichenketten übergeben. Da es keine Vorgabe ist, Kleinbuchstaben in dem Passwort zu verwenden, liefert auch die Zeichenkette ohne einen Kleinbuchstaben `true` zurück. In dem ersten Test `fun password validation - junit()` werden die Rückgabewerte mit Hilfe der JUnit Bibliothek überprüft. Die darin enthaltene Klasse `Assert` bietet die hier verwendeten Funktionen `assertTrue()` und `assertFalse()` an.

Weitere Funktionen sind zum Beispiel: assertArrayEquals(), assertEquals(), assertNotEquals(), assertNotNull(), assertNotSame() und assertNotNull().[32] Eine Bibliothek, welche verwendet werden kann, um die Tests lesbarer zu gestalten, ist die Bibliothek "Truth". Diese wurde parallel zu JUnit entwickelt und basiert auf derselben Funktionsweise. Einige der Unterschiede sind in der folgenden Tabelle zu sehen und werdem in der zweiten Funktion `password validation - truth()` verwendet.

JUnit	Truth
assertEquals(b, a)	assertThat(a).isEqualTo(b)
assertTrue(c)	assertThat(c).isTrue()
assertTrue(d.contains(a))	assertThat(d).contains(a)
assertTrue(d.contains(a) && d.contains(b))	assertThat(d).containsAtLeast(a, b)
assertTrue(d.contains(a) || d.contains(b))	assertThat(d).containsAnyOf(a, b)

Tabelle 3.1: Unterschiede zwischen JUnit und Truth

Da die Funktionen sich in puncto der Lesbarkeit sehr ähnlich sind und die Bibliothek Truth im gegensatz zu JUnit in diesem Fall nur aus optischen Gründen verwendet wird, wird diese in zukünftigen Tests nicht verwendet.

3.3.2 Modultest und ViewModels

Das Testen der Module in einem ViewModel erzeugt jedoch Komplikationen, welche mit der Verwendung des Entwurfsmusters "Model View ViewModels" zusammenhängt.

MVVM

MVVM steht für das Entwurfsmuster "Model View ViewModel" und ist eine Variante des Model-View-Controller-Musters. Verwendet wird dies, um die Verbindung zwischen der Logik und Benutzeroberfläche zu trennen. Die ViewModel-Klasse dient hierbei zum Speichern und Verwalten von Daten, welche für die Benutzeroberfläche notwendig sind. Zudem überstehen Daten den Lebenszyklus von `onCreate` bis `onDestroy` sowie die Drehung des Displays. Zudem können mehrere Fragments auf dasselbe ViewModel

zugreifen. Für den Login- und Anmeldeprozess benötigte Daten werden in diesem Projekt in dem `LoginViewModel` gehalten und aktualisiert. Bis die benötigten Daten von dem Server empfangen sind, werden hinterlegte Daten angezeigt. Dies ermöglicht die Benutzeroberfläche darzustellen, ohne einen Ladeprozess anzeigen zu müssen. Mithilfe eines Listeners können zudem neu empfangene Daten direkt auf der Oberfläche angezeigt werden. [23]

Wie in der Abbildung 3.2 zu sehen, stellt der Teil des Views in diesem Projekt beispielhaft das `LoginFragment` dar. Dieses verwendet wiederum das `LoginViewModel`, um die Daten der Benutzeroberfläche zu halten. Das `HStoreViewModel` erbt von dem `ViewModel`, welches wiederum eine von Android bereitgestellte Klasse ist. Zudem ist es der Vorgänger des `LoginViewModel`. Das LoginViewModel, HStoreViewModel und ViewModel befinden sich in dem Entwurfsmuster in dem Bereich ViewModel. Sowohl das LoginViewModel als auch das HStoreViewModel verwenden den Store, welcher sich mit dem HStore in dem `Model` befindt und die Brücke zum Backend darstellt. Die Verbindung zu dem Server wird aufgebaut, sobald der Store initialisiert wird. Auf diese Weise können die Vorteile des MVVM ausgenutzt werden, führt jedoch zu einer tiefen Implementierung des HStores in dem Projekt.

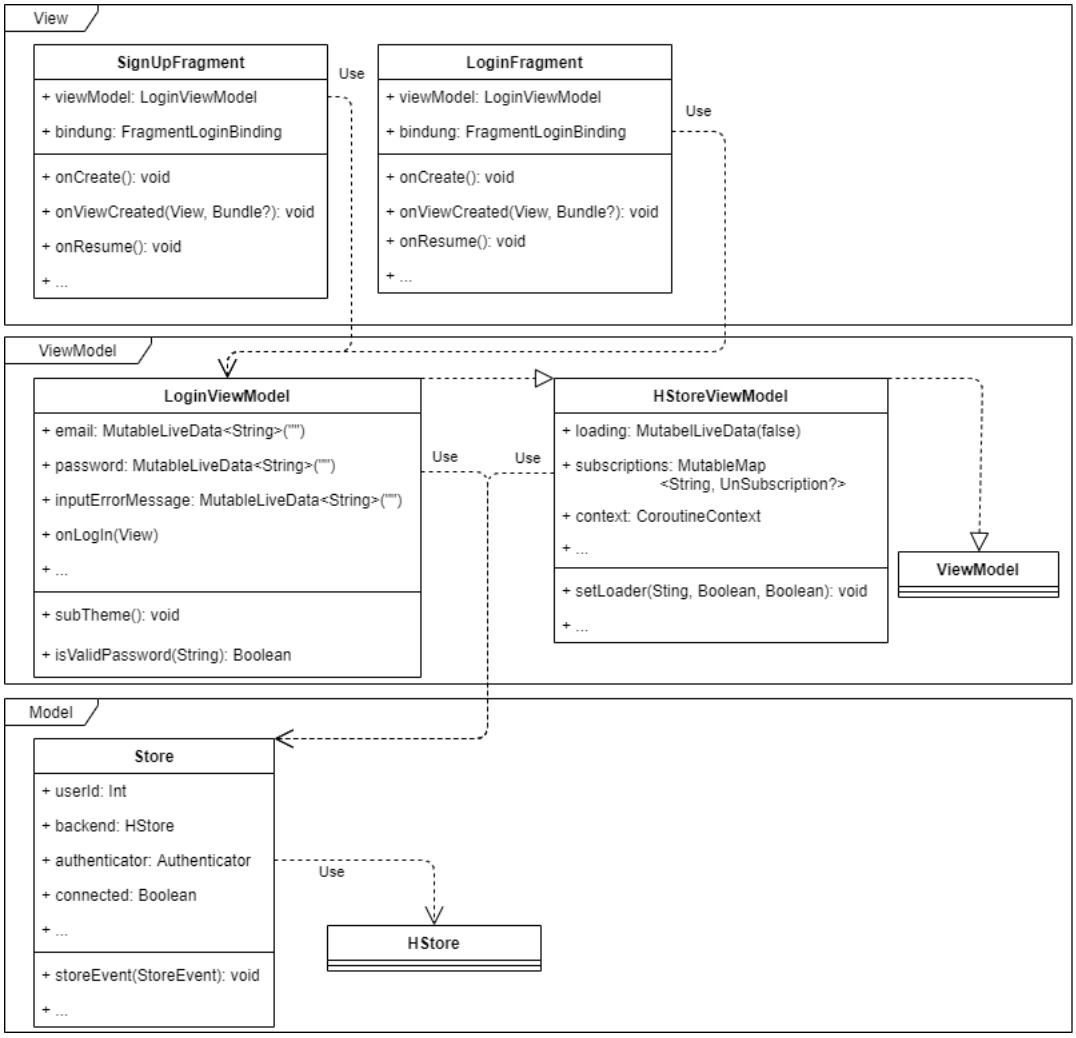


Abbildung 3.2: MVVM - HStore und LoginViewModel

Problematik

Die Klassen `HStoreViewModel` und `HStore` befinden sich in der eigens entwickelten Bibliothek, welche in `build.gradle` eingebunden wird. Die Bibliothek ist in C++ geschrieben und wird in Android-x86 kompiliert. Dies führt dazu, dass keine kompilierte Java-Version der Bibliothek existiert. Da Modultest in diesem Projekt auf einem macOS und keinem Android System ausgeführt werden, führt dies zu dem folgenden Fehler.

```
java.lang.UnsatisfiedLinkError: no native-lib in java.library.path:
 [/Users/simonborho/Library/Java/Extensions, /Library/Java/Extensions, /Network/Library/Java/Extensions,
  /System/Library/Java/Extensions, /usr/lib/java, .]

    at java.base/java.lang.ClassLoader.loadLibrary(ClassLoader.java:2670)
    at java.base/java.lang.Runtime.loadLibrary0(Runtime.java:830)
    at java.base/java.lang.System.loadLibrary(System.java:1873)
    at hamburg.appbase.lib.hstorenativeinternal.HStoreBridge.<clinit>(SourceFile)
    at hamburg.appbase.lib.hstorenative.HStore.<init>(SourceFile)
    at hamburg.appbase.pwc.awarenessapp.data.Store.<clinit>(Store.kt:124)
    at hamburg.appbase.pwc.awarenessapp.viewModels.LoginViewModelTest.setUp(LoginViewModelTest.kt:29)
```

Abbildung 3.3: Linkerror

Um dies zu beheben muss die Bibliothek in Java kompiliert werden, was aus technischer Sicht möglich ist. Im Rahmen dieses Projektes wird dies jedoch aus wirtschaftlichen Gründen nicht umgesetzt. Zudem ist es durch die Verwendung des MVVM-Musters und der tiefen Implementierung des HStore in dem Projekt nicht möglich, die Verwendung der Bibliothek zu umgehen. Somit erübrigt sich der erste Ansatz, Modultests für ein ViewModel zu schreiben. Es besteht jedoch die Alternative, die Modultests in `pwc/app/src/androidTest/java/x` anzulegen. Das führt dazu, dass die Anwendung vor dem Ausführen des Tests kompiliert wird und somit auf die Bibliothek zugreifen kann. Einer der daraus resultierenden Nachteile ist, dass für die Modultests ein Emulator gestartet werden muss. Zudem wird die Vorgabe von Android, dass Modultests sich in `pwc/app/src/test/java/x` befinden sollten, missachtet. Dies führt dazu, dass die Abdeckung des Codes durch Modultests nicht mehr mit dem internen Programm von Android Studio angezeigt wird. Da die Vorteile jedoch die Nachteile überwiegen, wird dieser Ansatz gewählt und umgesetzt.

LoginViewModelTest

Die Modultests für die ViewModels werden wie eingangs erläutert in dem Verzeichnis `pwc/app/src/androidTest/java/x` angelegt. Die zu testende Klasse `LoginViewModel` erbt von dem `HStoreViewModel` und enthält die MutableLiveDaten email, password, inputError und inputErrorMessage. In dem `fragment_login.xml`, welchem das viewModel zugewiesen ist, wird dem LoginButton `app:onClicked=@{viewModel.onLogIn}` zugewiesen. Wird der Button betätigt, werden der Variablen `email` und `password` aus den MutablenLiveDaten gelesen und zwischengespeichert. Anschließend wird überprüft, ob in beiden Variablen ein Inhalt

vorhanden ist. Wenn dies nicht der Fall ist, wird eine passende Nachricht in inputErrorMessage geschrieben und der inputError auf "true" gesetzt. Handelt es sich um valide Werte, werden diese an den `Store.login(Store.LoginData(email, password))` übergeben. Passen die übergebenen Werte zu einem Nutzer, wird `Success` zurückgegeben und der Nutzer ist eingeloggt. Andernfalls wird ein Fehler angezeigt.

Um diese Funktionalität zu testen, wird zunächst die Funktion `setUp` mit `Before` gekennzeichnet und somit vor jeder Testfunktion aufgerufen. Darin wird der `Store, internContext, customView` und das `loginViewModel` mit Hilfe von MockK gemockt. Der customView liefert als context immer den internContext zurück.

```
@RunWith(AndroidJUnit4::class)
class LoginViewModelUnit {

    private lateinit var loginViewModel: LoginViewModel
    private lateinit var internContext: Context
    private lateinit var customView: View

    @Before
    fun setUp() {
        MockKAnnotations.init(this)
        mockkObject(Store)

        internContext = InstrumentationRegistry.getInstrumentation()
            .targetContext
        loginViewModel = spyk(LoginViewModel())
        customView = mockkClass(View::class)
        every { customView.context } returns internContext

    }
```

Quellcode 3.2: LoginViewModel setUp()

Um dieselben Schritte nicht bei jedem Klick auf den Loginbutton ausführen zu müssen, wird die Hilfsfunktion `setAndClickLogIn(email, password, error, notification)` eingefügt. In dieser wird zunächst in dem LoginViewModel die übergebene E-Mail und das Passwort gespeichert. Da die Funktion onClicked eine View mit einem Context benötigt, wird onLogIn folgendermaßen aufgerufen: `loginViewModel.onLogIn.onClicked(customView)`. Anschließend wird mit `assertThat` überprüft, ob die im LoginViewModel hinterlegte inputErrorMessage und der inputError den zuvor übergebenen Werten entsprechen.

```
private fun setAndClickLogIn(email: String?, password: String?,
                             error: Boolean?, notification: String?) {
    loginViewModel.email.postValue(email)
    loginViewModel.password.postValue(password)

    loginViewModel.onLogIn.onClicked(customView)

    assertThat(loginViewModel.inputErrorMessage.value)
        .isEqualTo(notification)
    assertThat(loginViewModel.inputError.value).isEqualTo(error)
}
```

Quellcode 3.3: LoginViewModel setAndClickLogin()

In der Testfunktion `onLogInClickedEmptyNull` werden alle Kombinationen aus einem leeren String und null für die E-Mail und das Passwort an die gezeigte Funktion `setAndClickLogin(email, password, error, notification` übergeben. Der erwartete Wert für die Variable `inputError` ist hierbei immer `false` und die `inputErrorNotificaton` entspricht der in `R.string.error_message_email_empty` oder `R.string.error_message_password_empty` hinterlegten Zeichenkette.

Um die zwei Rückgabewerte des Servers `Store.AuthResult.Success` und `Store.AuthResult.Error(Sting)` zu überprüfen, wird zunächst in der Funktion `onLogInClickedSuccErr` der erfolgreiche Fall gemocket. Hierzu wird festgelegt, dass bei einer Übergabe mit den Werten `email=testmail@appbase.de, passwort=pwTes123` der `Store.AuthResult Success` zurück gibt. Danach folgt die Überprüfung mit Hilfe der Funktion `setAndClickLogIn()`. Anschließend wird für den selben Übergabewert definiert, dass der Server einen Fehler liefert und anschließend mit der Funktion `setAndClickLogIn` überprüft. Ist auch dieser Test bestanden, ist die gesamte Funktionalität überprüft.

```
@Test
fun onLogInClickedSuccErr() {
    coEvery {
        Store.login(
            Store.LoginData("testmail@appbase.de", "pwTes123*"))
    } returns Store.AuthResult.Success

    setAndClickLogIn(
        email = "testmail@appbase.de", password = "pwTes123*", error = false,
        notification = ""
    )

    coEvery {
```

```
            Store.login(
                Store.LoginData("testmail@appbase.de", "pwTes123*"))
        } returns Store.AuthResult.Error("Error")

        setAndClickLogIn(
            email = "testmail@appbase.de", password = "pwTes123*", error = true,
            notification = internContext.getString(R.string.login_error)
        )
    }
```

Quellcode 3.4: LoginViewModel onLoginClickedSuccErr()

Nach jeder Testfunktion wird die Funktion `cleanUp()` und somit `unmockAll()` aufgerufen, um die gemockten Daten in den jeweiligen Originalzustand zu bringen.

4 Integrationstests

Dieses Kapitel behandelt die Grundlagen und Anwendung der Integrationstests in dem Projekt. Hierzu wird das LoginFragment beispielhaft getestet. Zudem wird genauer auf das Framework Espresso eingegangen.

4.1 Grundlagen

Die Integrationstests befinden sich in der Testpyramide an mittlerer Stelle und sollten in etwa 20 Prozent der gesamten Tests ausmachen. Zudem werden diese im Gegensatz zu Modultests auf einem realen Gerät oder Emulator ausgeführt. Sie dienen zum validieren der Zusammenarbeit zwischen zusammenhängender Gruppen, welche gegebenenfalls unabhängig voneinander entwickelt wurden. Zudem kann geprüft werden, ob ein System aus mehreren Modulen wie erwartet funktioniert.[37] Integrationstests haben im Vergleich zu den Modultests einen großen Umfang und werden daher mit `@MediumTest` deklariert. Die Zeit, welche benötigt wird, um den Test auszuführen, sollte hierbei fünf Minuten nicht überschreiten. Im Verlauf des Testens werden sie nach den Modultests ausgeführt und sollten spätestens nach dem Fertigstellen eines Elementes oder Tickets durchgeführt werden. In einer Android-Anwendung kann so die Interaktion zwischen einer Ansicht und einem Ansichtsmodell geprüft werden.[25] Ein Beispiel hierfür ist das Testen eines Fragment-Objektes, das Validieren eines XML-Layout oder die Auswertung der Anbindung eines ViewModels. Durch die Verwendung von Espresso können Aufgaben synchronisiert ausgeführt werden, wodurch Aktionen auf der Oberfläche ausgeführt oder zu einem bestimmten Element in einer RecyclerView gescrollt werden kann. Zudem werden auch bei Integrationstests Test-Doubles verwendet, um die benötigten Komponenten unabhängig von dem Gesamtsystem testen zu können. Zudem kann ein Fragment mit einem vordefinierten Intent initialisiert werden. Im nächsten Schritt soll das LoginFragment mit den dazugehörigen XML und ViewModel überprüft werden.[29]

4.2 LoginFragment Integrationstests

Das `LoginFragment` ist das erste Fragment mit welchem der Nutzer interagieren kann. Integrationstests werden daher in dieser Arbeit anhand des LoginFragments dargestellt. Dieses steht exemplarisch für alle Fragments der Anwendung. Es beinhaltet die einzelnen Komponenten des Entwurfsmusters MVVM. Dazu gehört das `LoginFragment.kt`, `LoginViewModel.kt`, `fragment_login.xml` sowie der `Store` und `HStore`.

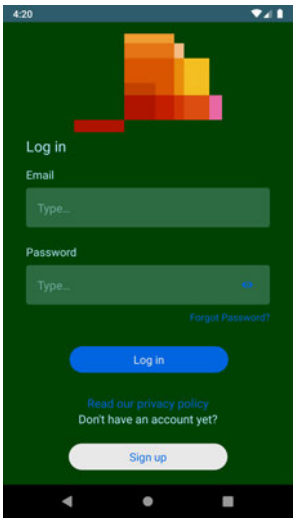


Abbildung 4.1: Login Fragment

In der Funktion `@Before setUp()` wird wie bei den Modultests zunächst `MockKAnnotations.init(this)` ausgeführt und der Variablen `instrumentationContext = InstrumentationRegistry.getInstrumentation().targetContext` zugewiesen. Durch den targetContext kann der Test auf die echten Ressourcen der App zugreifen, welche wiederum benötigt werden, um zum Beispiel auf die hinterlegten Schriftgrößen zugreifen zu können. Anschließend wird eine Mockklasse des HStores erzeugt und dem Spyk des LoginViewModels ein Theme zugewiesen. Anschließend soll das Fragment mit Hilfe von `launchFragment<LoginFragment>()` gestartet werden.

4.2.1 LaunchFragment

Um ein gesamtes Fragment zu testen, wird es mit Hilfe von `launchFragment<LoginFragment>()` auf einem Emulator oder realen Gerät gest-

artet. Jedoch kommt es durch die von Kotlin bereitgestellte Zuweisung des ViewModels `LogInViewModel by viewModels()` zu einem Fehler beim Starten des Tests.

```
java.lang.IllegalStateException: Can't access ViewModels from
detached fragment at
androidx.fragment.app.Fragment.getViewModelStore(Fragment.java:365)
```

Abbildung 4.2: LunchFragment Error

Um diesen zu beheben, ist eine Änderung in dem originalen Code notwendig. Die Variable `binding` wird mit der Notation `@VisibleForTesting(otherwise = VisibleForTesting.PRIVATE)` für die Tests sichtbar gemacht. Zudem wird `by viewModels()` durch `by provideViewModel` ersetzt.

```
private val viewModel: LogInViewModel by viewModels()
private lateinit var binding: FragmentLoginBinding

val viewModel: LoginViewModel by provideViewModel()
@VisibleForTesting(otherwise = VisibleForTesting.PRIVATE)
lateinit var binding: FragmentLoginBinding
```

Quellcode 4.1: Änderung des Orginalcodes

```
class OverridableLazy<T>(var implementation: Lazy<T>) : Lazy<T> {
    override val value
        get() = implementation.value

    override fun isInitialized() = implementation.isInitialized()
}

object HelperViewModel {
    inline fun <reified VM : ViewModel> Fragment.provideViewModel(
        noinline factoryProducer: (() -> ViewModelProvider.Factory)? = null
    ): Lazy<VM> =
        OverridableLazy(activityViewModels(factoryProducer))
}
```

Quellcode 4.2: ProvideViewModel

Um dem Fragment nun in einem Test ein gemocktes ViewModel übergeben zu können, wird die Hilfsfunktion `launchLoginFragment(ViewModel)` implementiert. In ihr wird, wie eingangs erwähnt, das Fragment mit Hilfe der Funktion

`launchFragmentInContainer<LoginFragment>()` gestartet und das ViewModel mit dem customViewModel und die Variable `binding.viewModel` mit dem customViewModel ersetzt.

```
fun lunchLoginFragment(customViewModel: ViewModel): FragmentScenario<*> {
    return launchFragmentInContainer<LoginFragment>().onFragment { f ->
        f.apply {
            replace(LoginFragment::viewModel, customViewModel)
            this.binding.viewModel = customViewModel as LoginViewModel
        }
        Thread.sleep(500)
    }
}
```

Quellcode 4.3: Lunch Fragment

So kann in dem eigentlichen Integrationstest das Fragment mit `scenario = launchLoginFragment(loginViewModel)` gestartet und das gemockte loginViewModel übergeben werden.

4.2.2 Espresso

Bei Espresso handelt es sich um ein Testframework für Android, welches das Schreiben der Tests erleichtern soll. Es kann verwendet werden, um Elemente in der UI zu finden und anschließend mit diesen zu interagieren. Gleichzeitig verhindert das Framework den direkten Zugriff auf die Aktivitäten und Sichten der Anwendung und stellt sicher, dass die Activity vor dem Test gestartet wird. Um Tests in Espresso ausführen zu können, ist es notwendig, die Animationen in den Entwickleroptionen auszuschalten. Hierzu wird der Wert der `Animationsmaßstab 1x` auf `Animation aus` gestellt. Dies ist für die Punkte `Maßstab für Fensteranimation`, `Maßstab für Übergangsanimation` und `Maßstab für Animatorzeit` notwendig. Espresso besteht im Wesentlichen aus den drei Komponenten ViewMatcher, ViewActions und ViewAssertions.[21]

ViewMatcher

Der ViewMatcher ermöglicht das Auffinden von Views in der aktuellen View-Hierarchie. Hierfür wird der Funktion `onView()` ein ViewMatcher wie zum Beispiel `withId(R.id.textView)` übergeben. Anschließend wird aus einer Liste, welche alle

Elemente der angezeigten View enthält, das passende Element ausgewählt. Um dies genauer beschreiben zu können, kann man mit `allOf()` weitere Parameter angeben. Als weitere Spezifikationen können Funktionen wie `withText()`, `withContentDescription()` oder `withParent(withId()))` verwendet werden. Dies ist zum Beispiel in Listen notwendig, da jedes Element dieselbe ID enthält. Zudem besteht die Möglichkeit, einen eigenen Matcher zu schreiben. So kann überprüft werden, ob es sich bei der gegebenen Schriftgröße um die gewünschte handelt. Dies wird mit Hilfe des CustomMatcher `withTextViewSizeMatcher()` realisiert. Hierfür wird der Funktion die Textgröße als Int-Wert übergeben und ein BoundedMatcher zurückgegeben. Innerhalb des Matchers wird die Beschreibung überschrieben, sodass die Ausgabe in der Konsole aussagekräftig ist. Mit `override fun matchesSafely()` wird der gegebene und der erwartete Wert verglichen und das Ergebnis als Boolean zurückgegeben. Handelt es sich hierbei um einen falschen Wert, wird die Fehlermeldung ausgegeben, welche in der Funktion `describeMismatch()` überschrieben wird. Um den Matcher verwenden zu können, wird dieser wie folgt eingebunden:
`title.check(matches(withTextViewSizeMatcher(10)))`.

```
fun withTextViewSizeMatcher(textSize: Int): Matcher<View?>? {
    return object : BoundedMatcher<View?, TextView>(TextView::class.java) {

        override fun describeTo(description: Description) {
            description.appendText("with textSize: $textSize")
        }

        override fun describeMismatch(item: Any, description: Description) {
            description.appendText("string not match").appendValue((item as Int))
        }

        override fun matchesSafely(item: TextView?): Boolean {
            return item?.textSize?.toInt() == textSize
        }
    }
}
```

Quellcode 4.4: withTextViewSizeMatcher

ViewActions

Anschließend kann auf diesem mit Hilfe von ViewActions eine oder mehrere Aktionen mittels `perform()` ausgeführt werden. Bei Aktionen handelt es sich um Ausführungen wie einen Klick `click()`, das Scrollen zu dem jeweiligen Element `scrollTo()` oder

das Eingeben eines Textes `typeText()`. Aktionen können zudem verkettet und somit nacheinander mit `perform(scrollTo(), click())` ausgeführt werden.

ViewAsserts

Ist die Aktion ausgeführt, kann das Ergebnis mit Hilfe von ViewAssertions und der Funktion `check()` überprüft werden. Die am häufigsten verwendete Assertion ist `matches()`. So kann mit `matches(withText(""))` überprüft werden ob der gegebene Text mit dem erwarteten übereinstimmt. Zudem können eigene Assertions wie `withTextViewColorAssertions` definiert werden. Der Funktion wird die Farbe des Textes und gegebenenfalls eine Hintergrundfarbe übergeben. Anschließend wird überprüft, ob ein View gefunden werden kann und ob es sich bei diesem um eine TextView handelt. Ist dies nicht der Fall, wird eine passende Fehlermeldung ausgegeben. Danach wird mit `assertThat()` die Übereinstimmung des übergebenen und tatsächlichen Wertes überprüft.

```
fun withTextViewColorAssertions(textColor: Int, backgroundColor: Int = -1):
        ViewAssertion {
    return ViewAssertion { view, noViewFoundException ->
        if (noViewFoundException != null) {
            throw noViewFoundException
        }
        if (view !is TextView) {
            throw IllegalStateException("The asserted view is " +
                    "not type of TextView")
        }

        MatcherAssert.assertThat("TextView text color",
            view.currentTextColor, CoreMatchers.equalTo(textColor))
        if (backgroundColor != -1) {
            MatcherAssert.assertThat("EditText background color",
                (view.background as ColorDrawable)
                    .color, CoreMatchers.equalTo(backgroundColor))
        }
    }
}
```

Quellcode 4.5: withTextViewColorAssertions

4.2.3 Anwendung

Nachdem eine Übersicht über Espresso gegeben und das Fragment gestartet ist, können die einzelnen Elemente überprüft werden. So wird zunächste das zu testende View mit `onView(id)` in der Variablen `passwordInput` gespeichert. Bei dem passwordInput handelt es sich um ein eingebundenes `ConstraintLayout`, welches den Titel (`TextView`) und einen Container (`ConstraintLayout`) enthält. Dies beinhaltet wiederum ein Eingabefeld (`EditText`) und situationsbedingt eines von drei Symbolen (`ImageView`).

Abbildung 4.3: Passwort Eingabefeld

Im nächsten Schritt wird `passwordInput.perform(scrollTo())` ausgeführt. Hierbei wird auf einer ScrollView zu dem angegebenen Element gescrollt. Ist das Element aufgrund der Position im Layout nicht sichtbar, kann keine weiter Aktion auf diesem ausgeführt werden und ein Fehler wird geworfen. Anschließend wird mit `check(matches(isDisplayed()))` überprüft, ob das Element sichtbar ist. Mit Hilfe von `checkBasics(Int, String, ICON)` können die grundlegenden Werte des Eingabefeldes überprüft werden. Hierzu wird der Funktion die parentId, der Titel und gegebenenfalls das Symbol übergeben. Danach werden die Werte mit den dazugehörigen Matchern geprüft.

```
val passwordInput = onView(ViewMatchers.withId(R.id.password_input))
passwordInput.perform(scrollTo())
passwordInput.check(matches(isDisplayed()))
InputField.checkBasics(
    parentId = R.id.password_input, title = "Password",
    icon = ICON.PASSWORD)
InputField.checkColor(
    parentId = R.id.password_input, titleColor = mockTheme.mainText,
    contentColor = mockTheme.mainText, backgroundColor = mockTheme.mainInput)

}
```

Quellcode 4.6: Passwort Eingabefeld Testausschnitt

Der Aufbau `withParent(withParent(withId(parentId)))` ist vonnöten, da der Titel sich aus der Sicht des angezeigten Fragments innerhalb einer Verschachtelung aus mehreren Views befindet.

```
fun checkBasics(parentId: Int, title: String, icon: ICON = ICON.NONE) {
    onView(allOf(withId(R.id.titleTextView), withParent(withId(parentId))))
        .check(matches(isDisplayed()))
        .check(matches(withText(title)))

    onView(allOf(withId(R.id.editText), withParent(withParent(withId(parentId)))))
        .check(matches(isDisplayed()))
        .check(matches(withHint(R.string.hint_type)))

    checkIconState(parentId, icon != ICON.NONE, icon)
}
```

Quellcode 4.7: CheckBasics Funktion

In der Testfunktion `loginDifferentInput()` werden zunächst wie in dem Modultest die Rückgabewerte des Servers für einen Login gemockt. Dann werden diese mit ViewActions in die dafür vorgesehenen Felder eingetragen und bestätigt. Ist das Einloggen nicht erfolgreich, wird eine Fehlermeldung angezeigt. Die dazugehörige View befindet sich in dem `activity_main.xml`, welches die unterste Ebene der Fragments darstellt und ist nur über die `MainActivity` zugänglich. So kann erst in den Integrationstests geprüft werden, ob die Fehlermeldung wie erwartet angezeigt wird. Der Aufbau des gesamten Integrationstests für das LoginFragment ist im Anhang unter dem Abschnitt A.1 zu finden. Der Aufbau für die jeweiligen Elemente ähnelt hierbei dem des Eingabefeldes für das Passwort.

5 Tests der Benutzeroberfläche

Dieses Kapitel behandelt die Tests der Benutzeroberfläche, wie diese durchgeführt werden können und zu welchen Komplikationen es bei diesen kommen kann. Hierfür werden in dieser Arbeit zwei, in der Umgebung von Android häufig genutzte, Testkonzepte beleuchtet, welche das Testen der Benutzeroberfläche ermöglichen.

Obwohl Tests der Benutzeroberfläche nur etwa 10 Prozent der gesamten Tests ausmachen sollten, handelt es sich bei diesen um einen wesentlichen und wichtigen Teil des Testens. Hierbei kann eine zuvor definierte User Story durchgeführt werden, welche den Benutzer durch mehrere Module und Funktionen führt. Zudem kann diese Art der Tests Engpässe in der Anwendung aufzeigen, da sie sehr nah an der tatsächlichen Verwendung der Anwendung liegt. Typischerweise wird diese Art der Tests auf einem Emulator oder realen Gerät manuell oder automatisiert ausgeführt. Bei dem manuellen Testen werden die zuvor definierten User Storys oder einzelne Funktionalitäten der Anwendung von einer realen Person durchgeführt und abgenommen. Hierbei kann eine reale Rückmeldung über die Funktionalität sowie die Leistung und Nutzbarkeit gegeben werden. Automatisierte Tests können häufiger ausgeführt werden und sind somit auf die Dauer gesehen kostengünstiger. Jedoch können keine realen Bedingungen, wie die Sonneneinstrahlung, berücksichtigt werden. Zudem handelt es sich bei den Tests der Benutzeroberfläche um fragile Tests, da es selbst durch kleine Änderungen an der Oberfläche zum Misslingen der Tests kommen kann.[42] Die wichtigsten Aspekte der Tests der Benutzeroberfläche sind hierbei die Funktionalität der Anwendung, deren visuellen Gestaltung und Benutzerfreundlichkeit sowie die Leistung und dazugehörige Engpässe.

Die Fragilität der Tests der Benutzeroberfläch wird in der Studie “Automated Mobile UI Test Fragility” genauer beleuchtet. Das Ziel der Studie besteht darin, eine explorative Bewertung der Ursachen zu finden und deren Ausmaße zu bewerten. Hierfür werden für die Testframeworks Espresso, UIAutomator, Selendriod, Silk Mobile und Sikuli GUI Automation Tool jeweils ein kleiner Test-Suite entwickelt, um die Veränderung zu erkennen, welche durch kleine Änderungen an der Oberfläche hervorgerufen werden. Die Tools

können in die Gruppe der auf Code basierenden und der graphischen Tests unterteilt werden. Die Auswertung der jeweiligen Tests ergab, dass bei den auf Code basierenden Tests bis zu 75 Prozent und bei der Bilderkennung bis zu 100 Prozent angepasst werden müssen. Die Hauptursachen der Anfälligkeit liegen hierbei bei der Anpassung von Bezeichnern, Texten oder Grafiken, sowie das Entfernen oder Verschieben von Elementen. Zudem führt die Anpassung des Aktivitätsfluss sowie die Variation der Zeit, welche zum Ausführen benötigt wird, zum Scheitern der Tests.[42] Aufgrund der Resultate der Studie werden in dieser Arbeit von den genannten und auf codebasierten Tools Espresso und der UIAutomater sowie ein automatisierter Klicker angewendet.

5.1 Test einer User Story

Eine Möglichkeit, die Funktionalität einer Anwendung zu testen, ist das Entwickeln eines Tests für jede User Story des Projektes. Bei einer User Story handelt es sich um die Anforderungen an ein Produkt oder Service aus der Sicht eines Nutzers und ist ein Werkzeug der agilen Softwareentwicklung. Bei der Erstellung einer User Story wird die Erwartung der Nutzer festgehalten und nicht wie diese Umgesetzt werden. Dabei folgen sie dem entsprechendem Aufbau: Als [Rolle] möchte ich [Funktion] um [Nutzen]. Die Kriterien, um eine User Story zu bewerten, sind, dass diese unabhängig von anderen User Stories ist und es sich um keinen unumstößlichen Vertragstext handelt. Zudem sollte sie nützlich sein, einen Mehrwert liefern und der Aufwand, welcher benötigt wird, um diese zu realisieren, muss abschätzbar sein. Des Weiteren ist eine gute User Story in einem Sprint realisierbar und somit klein und erfüllt die Vorgaben testbar zu sein.[38] Im Folgenden wird im ersten Schritt eine solche mit Hilfe des Tools “Espresso” realisiert.

5.1.1 Espresso

Das Tool Espresso kann nicht nur für die Integrationstests verwendet werden, sondern bietet auch die Möglichkeit, Tests der Benutzeroberfläche zu erstellen. Die Vorgehensweise entspricht hierbei den im bereits aufgezeichneten Kapitel 5.2.2 aufgezeigten. Der Unterschied ist hierbei, dass nicht ein Fragment oder zusammenhängende Komponenten getestet werden, sondern nach einem vordefinierten Durchlauf durch die Anwendung navigiert und interagiert wird. Es können ebenfalls Daten gemockt werden. Um jedoch eine möglichst realistische Umgebung zu schaffen, wird hierauf weitestgehend verzichtet

und die Daten von einem echten Server bezogen. Um eine immer gleiche Testumgebung zu erhalten, ist es möglich, einen Testserver einzubinden, welcher vor jedem Test neu hochgefahren wird und somit immer dieselbe Ausgangssituation bereitstellt.[39]

In diesem Projekt wird als Beispiel für das Testen der User Story der Bereich des LoginFragments sowie einige Funktionalitäten, welche mit der Registrierung zusammenhängen, überprüft. Um diese durchzuführen, wird zunächst die Umgebung vorbereitet. Hierzu wird die Activity zugewiesen und in `setup()` die benötigten Komponenten gemockt. Des weiteren wird der `DeviceHelper` gestartet, der das Gerät mindestens einmal alle zwei Sekunden dreht. Nach den Tests wird die Funktion `cleanUp()` ausgeführt. Darin wird der `LifecycleManager` und `DeviceHelper` beendet und die gemockten Objekte zurückgesetzt.

```
@get:Rule
val activeRule = ActivityScenarioRule(MainActivity::class.java)

@Before
fun setup() {
    MockKAnnotations.init(this)
    mockkObject(Store)
    mockkClass(StoreEvents::class)
    DeviceHelper.initDeviceHelper(rotationTimeMS = 1000)
}

@After
fun cleanUp() {
    LifecycleManager.clear()
    DeviceHelper.cleanUp()
    unmockkAll()
}
```

Quellcode 5.1: Umgebung der Tests einer User Story

Die erste zu testende User Story lautet wie folgt: Als Nutzer möchte ich mich mit einer E-Mail-Adresse registrieren können, um einen Account zu erstellen. Wie in dem ersten Test wird die Anwendung gestartet. Anschließend soll ein neuer Account angelegt werden. Hierfür wird mit einem Klick auf den Registrieren-Button auf das Fragment zum Registrieren navigiert und die Felder “E-Mail”, “Passwort” und ıPasswort wiederholen” mit passenden Werten befüllt. Da fehlerhafte Werte und deren Auswirkung bereits in dem dazugehörigen Integrationstest überprüft wurden, ist dies nicht unbedingt notwendig, könnte jedoch ebenfalls an dieser Stelle wiederholt überprüft werden. Da in diesem Projekt kein Testserver vor dem Test hochgefahren wird und es den angegebenen Benutzer bereits gibt, wird auch hier der Rückgabewert des Servers für die Registrierung mit den

gegebenen Werten gemockt. Nach der Betätigung des Buttons für die Registrierung wird der Benutzer auf das SignUpInfoFragment weitergeleitet, welches Informationen über den weiteren Prozess der Registrierung anzeigt. Durch das Klicken auf den Anmelde-Button wird der Benutzer zurück zu dem LoginFragment geleitet, wobei die zuvor angegebene E-Mail-Adresse übernommen wird. Nach diesem Schritt wurde ein neuer Account angelegt und der Test gilt als bestanden.

```
@Test
fun registerTest() {
    onView(withId(R.id.textView)).check(matches(isDisplayed()))

    ButtonStyled.performClick(parentId=R.id.include_sign_up)
    InputField.checkSetContent(parentId=R.id.email_input,
            "test@appbase.hamburg")
    InputField.checkSetContent(parentId=R.id.password_input,
            "A1*awe3§")
    InputField.checkSetContent(parentId=R.id.password_input_repeat,
            "A1*awe3§")

    coEvery {
        Store.register(Store.RegisterData("test@appbase.hamburg",
            "A1*awe3§"))
    } returns Store.AuthResult.Success(1)

    ButtonStyled.performClick(parentId = R.id.include)
    onView(withId(R.id.headline)).check(withText("Sign up"))

    ButtonStyled.performClick(parentId = R.id.include_sign_up)
    onView(withId(R.id.textView)).check(matches(isDisplayed()))
}
```

Quellcode 5.2: Aufbau der ersten User Story

Die zweite zu testende User Story lautet: Als Nutzer möchte ich mich mit einem bestehenden Account einloggen können, um die Anwendung in vollem Umfang verwenden zu können. Hierzu wird wie in den vorherigen Tests die Anwendung gestartet. Nach dem das LoginFragment angezeigt wird, wird die E-Mail-Adresse, welche in dem vorherigen Test registriert wurde, eingegeben. Anschließend wird das Passwort eingegeben und durch einen Klick auf den Anmelde-Button werden die eingegebenen Daten zum Server gesendet, welcher diese bestätigt. Ist dies der Fall, wird man auf das AuthFragment weitergeleitet, in welchem ein Code eingegeben werden muss, den der Nutzer in einer E-Mail erhält. Hierbei handelt es sich um eine Zwei-Faktor-Authentifizierung, welche dem Schutz des Accounts dient. Auch dieser Wert wird manipuliert und nicht dem Server übergeben. Stattdessen wird dieser von einem Mock bestätigt und der Benutzer wird auf die Startsei-

te eines eingeloggten Benutzers weitergeleitet. Konnten diese Schritte ausgeführt werden, konnte zugleich die User Story durchgeführt werden und der Test gilt als bestanden.

```
@Test
fun loginTest() {
    InputField.checkSetContent(parentId = R.id.email_input,
            "test@appbase.hamburg")
    InputField.checkSetContent(parentId = R.id.password_input,
            "A1*awe3§")

    coEvery {
        Store.login(Store.LoginData("test@appbase.hamburg", "A1*awe3§"))
    } returns Store.AuthResult.Success(1)

    ButtonStyled.performClick(parentId = R.id.include)
    InputField.checkSetContent(parentId = R.id.auth_input, "ab123")

    coEvery {
        Store.twoFactorValidation(Store.
            TwoFactorData("test@appbase.hamburg","ab123"))
    } returns Store.AuthResult.Success(1)

    ButtonStyled.performClick(parentId = R.id.auth_button)
    onView(withId(R.id.gameView)).check(matches(isDisplayed()))
}
```

Quellcode 5.3: Aufbau der zweiten User Story

Da Espresso auf die Identifikationen der einzelnen Elemente zugreifen kann, ist es möglich, ein reales Gerät oder Emulator mit beliebigen Spezifikationen zu verwenden. Sollten jedoch in dem zu testenden Layout Änderungen vorgenommen werden, müssen auch die dazugehörigen Tests angepasst werden. Notwendig ist dies, sobald ein Element entfernt wird, die Identifikatoren sich ändern oder Interaktionen mit diesen eine andere Auswirkung beinhalten. Bei einer Umstrukturierung, bei welcher die bereits bestehenden Elemente bestehen bleiben, ist dies jedoch nicht notwendig. Ein Vorteil, welcher das Schreiben eines Tests für eine User Story mittels Espresso mit sich bringt, ist, dass die Umgebung an die jeweiligen Bedürfnisse angepasst werden kann, dies aber nicht notwendig ist. Zudem können Anpassungen oder Änderungen einfach realisiert werden, was jedoch manuell bewerkstelligt werden muss. Existieren Lücken oder Fehler in der User Story, werden diese von dem Test nicht aufgezeigt. Es besteht die Möglichkeit die Tests direkt in der Entwicklungsumgebung, in einer eigenen oder gemieteten Geräte-Cloud zu starten.

5.1.2 Espresso Test Recorder

Eine weitere Methode um eine User Story zu erstellen, welche ebenfalls mit dem Tool "Espresso" realisiert wird, ist die Aufnahme dieser mit Hilfe des "Espresso Test Recorders". Hierbei handelt es sich um ein Tool, welches von Android Studio bereitgestellt wird, um Tests der Benutzeroberfläche zu erstellen, ohne Testcode zu schreiben. Hierfür wird die Interaktion mit einem physischen Gerät oder Emulator aufgezeichnet, aus welchem anschließend der Test der Anwendung erzeugt wird. Dieser kann im darauffolgenden Schritt manuell bearbeitet und angepasst werden. Um einen Test aufzunehmen, wird zunächst in Android Studio über das Menü der Punkt "Record Espresso Test" und anschließend in dem Fenster "Select Deployment Target" ausgewählt und mit "OK" bestätigt. Im Anschluss wird das Projekt gebaut und installiert. Danach nimmt der Test die Interaktionen des Nutzers auf. Assertions können während der Interaktion mit der Oberfläche eingefügt werden. Ein TextView kann ausgewählt und hierbei angegeben werden, dass es sich um einen CustomTextView handelt und dieser die Zeichenkette "Log in" beinhaltet.

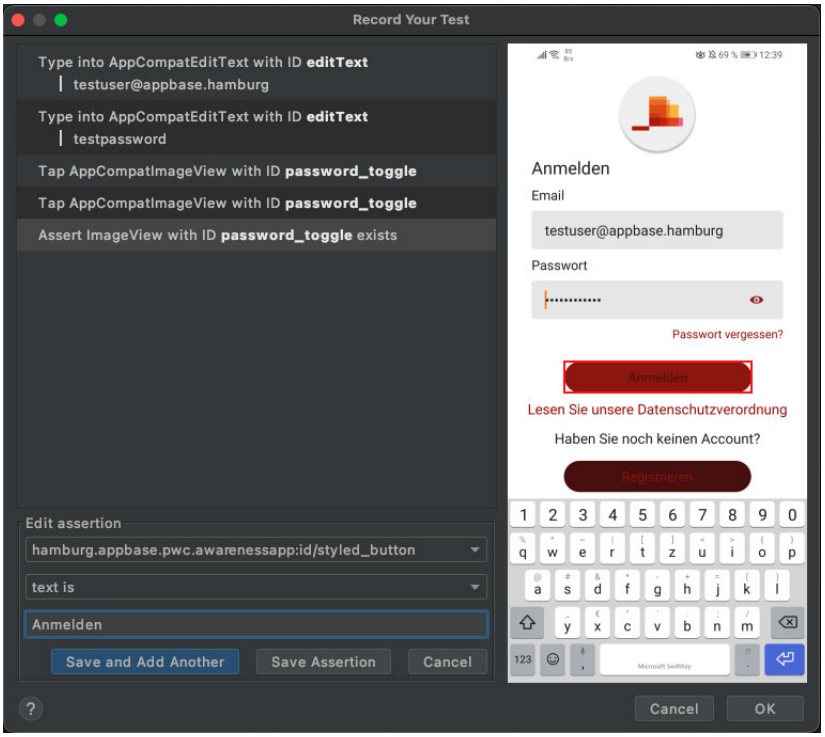


Abbildung 5.1: Espresso Test Recorder

Um den Test zu beenden wird, der Button “Complete Recording” betätigt und der Name des Tests kann geändert werden. Mit “Save” wird der Test in dem Verzeichnis `appname/java/androidTest/src/testName` gespeichert und automatisch geöffnet. Anschließend kann der erzeugte Code, wenn nötig, angepasst werden.[20] Mit dem Espresso Test Recorder ist es auch für das Personal möglich, einen Test einer User Story zu erzeugen, welches sich nicht mit der Entwicklung für Apps in Android beschäftigt. Kommt es zu Komplikationen während des Tests oder muss dieser im Nachgang angepasst werden, ist jedoch die Erfahrung eines Entwicklers vonnöten.

5.1.3 UI Automator

Einen Nachteil, welchen das Tool "Espresso" mit sich bringt, ist, dass es nur innerhalb der zu testenden Anwendung interagieren kann. Um den Zugriff auf das Gesamtsystem dennoch zu ermöglichen, kann das Tool "UI Automator" verwendet werden. Bei diesem handelt es sich um ein UI-Test-Framework, welches Anwendungen funktional, aber auch anwendungsübergreifend testen kann. Da der UI Automator, ebenso wie Espresso, auf Android Instrumentation basiert, können beide Tools miteinander kombiniert werden. Empfohlen wird hierbei, dass die Basis der Tests mittels Espresso geschrieben werden und die Interaktionen mit dem Betriebssystem mittels des Automator zu ergänzen. Mit Hilfe der GUI, welche mitgeliefert wird, kann die UI der Anwendung gescannt und analysiert werden um die geschriebenen Tests zu verfeinern. Die Verwendung des Tools bringt zudem weitere Vorteile mit sich. So kann während der Tests auf die Einstellungen des Gerätes zugegriffen, eine App vom Homescreen aus gestartet oder auch die Displaygröße abgegriffen werden. Des Weiteren ist es möglich das Gerät virtuell zu drehen, die Lautstärke anzupassen, einen der drei Buttons zur Navigation, Home, Menü und Zurück zu verwenden, einen Screenshot aufzunehmen oder mit Benachrichtigungen zu interagieren. Mit `device = UIDevice.getInstance(getInstrumentation())` kann das aktuell verwendete Gerät einer Variablen zugewiesen werden. Auf diesem können beispielhaft mittels `device.takeScreenshot()`, `device.getDisplayWidth()` oder `device.wakeUp()` die oben genannten Funktionalitäten verwendet werden. Anwendung findet der UI Automator in diesem Projekt im DeviceHalper Objekt. Durch den Aufruf der Funktion `DeviceHelper.initDeviceHelper()` kann ein automatisiertes und zufälliges Rotieren des Gerätes initialisiert werden. Die Zeitspanne zwischen den Rotationen liegt hierbei zwischen einer Sekunde und der mitgegeben Zeit in Sekunden. Im Zuge der Initialisierung wird zunächst, wie oben beschrieben, das Gerät zugewiesen. Anschließend wird die aktuelle Ausrichtung ausgelesen und in die entsprechend andere Richtung rotiert. Dieser Schritt wird ausgeführt, bis der Test abgeschlossen ist. In dem jeweiligen Test wird hierfür in `@After cleanUp()` die Funktion `DeviceHelper.cleanUp()` aufgerufen, welche wiederum den TimerTask beendet.

```
object DeviceHelper {
    private var currentPosition: Boolean = true
    private lateinit var device: UiDevice
    private var timerTask: TimerTask? = null
    private val playbackProgressTimer = Timer()

    fun initDeviceHelper(rotationTimeMS: Long = 5000) {
        device = UiDevice.getInstance(InstrumentationRegistry.getInstrumentation())
```

```
        if (timerTask == null) {
            timerTask = object : TimerTask() {
                override fun run() {
                    rotateDevice()
                }
            }
            playbackProgressTimer.schedule(timerTask, 1000, rotationTimeMS)
        }
    }

    fun cleanUp() {
        timerTask?.cancel()
        timerTask = null
    }

    private fun rotateDevice() {
        currentPosition = device.isNaturalOrientation
        if (currentPosition) {
            if (Random.nextBoolean()) {
                device.setOrientationLeft()
            } else {
                device.setOrientationRight()
            }
        } else {
            device.setOrientationNatural()
        }
    }
}
```

Quellcode 5.4: DeviceHelper

5.2 Monkeyrunner

Der "Monkeyrunner" ist ein von Android Studio bereitgestelltes Tool, welches das Steuern eines Emulators oder Android-Gerätes von außerhalb und mit Hilfe einer API ermöglicht. Die Programme, welche in Python geschrieben werden, ermöglichen das zu testende Paket zu installieren und anschließend auszuführen. Zudem können Eingaben über die Tastatur simuliert, Screenshots aufgenommen und Klicks auf dem Display ausgeführt werden. Im Vergleich zum "Monkey-Tool", welches mittels einer adb-Shell direkt auf dem Gerät oder Emulator pseudozufällige Eingaben eines Benutzers oder des Systems erzeugt, steuert der Monkeyrunner das Gerät oder Emulator von einer Workstation aus, indem spezifische Befehle und Ereignisse mit Hilfe der API übermittelt werden. Zudem ist es möglich, eine Sammlung von Tests auf einem oder mehreren Emulatoren oder physischen Gerä-

ten anzuwenden. Dies ermöglicht das gleichzeitige Starten und Verbinden mit Hilfe eines Programmes, welches anschließend mehrere Test auf dem jeweiligen Gerät ausführt. Zudem besteht die Möglichkeit, einen zuvor konfigurierten Emulator zu starten und diesen nach der Durchführung der Tests herunterzufahren. Das Monkeyrunner-Tool verwendet Jython, eine Implementierung von Python, welche Java als Programmiersprache verwendet. Dies ermöglicht die Verwendung der Syntax von Python, um auf Konstanten, Klassen und Methoden der API zuzugreifen und stellt eine einfache Interaktion mit dem Android-Framework. Das Tool ist in erster Linie für das Testen von Anwendungen und Geräten auf Funktions-/Framework-Ebene konzipiert.[22]

5.2.1 Anwendung

Um den einen Test mit dem Monkeyrunner zu starten, wird zunächst mit dem Befehl `MonkeyRunner.waitForConnection` das verbundene Gerät oder Emulator als `MonkeyDevice - Objekt` der Variablen `device` zugewiesen. Anschließend wird auf diesem das zu testende APK mit Hilfe des Befehls `device.installPackage(app.apk)` installiert. APK steht hierbei für das Android Package und ist ein Paketdateiformat, welches vom Android-Betriebssystem für die Distribution und Installation von Anwendungen verwendet wird. Um eine APK zu erstellen, wird das Projekt zunächst in Android-Studio kompiliert und in eine Container-Datei gepackt. Die APK umfasst alle Teile des Codes, die Ressourcen und Assets sowie Zertifikate und das Manifest. Ist die Datei erstellt, wird diese an einem zuvor definierten Ort und Namen abgelegt und kann der zuvor genannten Funktion übergeben werden.[18] Ist die Installation erfolgreich, liefert die Funktion einen `Boolean-Wert` zurück. Anschließend wird der Variablen `package` der Name des Paketes, welches im `AndroidManifest.xml` unter `package=""` zu finden ist, hinzugefügt. Dann wird die Activity einer Variablen zugewiesen. Hierbei ist darauf zu achten, dass der vollständige Pfad angegeben werden muss. Danach wird der Name der Komponente zusammengefügt `runComponent = package+/+activity`, um die Activity starten zu können: `device.startActivity(component=runComponent)`. Treten Fehler im Verlauf des Testens auf, werden diese mit Hilfe von `Logger.getLogger("com.android.chimpachat.adb.AdbChimpDevice")` aus dem Log ausgelesen und in die Errors eingetragen. Ist die Anzahl der Errors nach den Tests größer als null, können diese in der Console ausgegeben oder in einer Datei gespeichert werden. Anschließend kann der eigentliche Test gestartet werden. In dem zu

testenden Fall wird zunächst das Einloggen eines Benutzers simuliert. Hierfür kann mittels `device.touch(x, y, action)` ein Klick auf dem Gerät an den gegebenen Koordinaten ausgeführt werden. Mittels Konstanten des MonkeyDevice wie zum Beispiel `DOWN, UP oder DOWN_AND_UP` oder Methoden wie `drag, press oder touch` können Eingaben oder Aktionen an das Gerät weitergegeben werden. Um die Koordinaten zu erhalten, bietet Android in den Entwickleroptionen die Möglichkeit an, die aktuelle Position und Bewegung eines Klicks oder Bewegung anzuzeigen. Mit einem Klick in das Eingabefeld kann nun mit Hilfe von `device.type("test@mail.de")` die E-Mail-Adresse und das Passwort eingetragen werden. Anschließend folgt ein Ablauf, welcher sich beliebig oft wiederholen lässt. Hierbei wird im ersten Schritt zu dem Startpunkt, welcher in diesem Fall das `HomeFragment` ist, navigiert. Danach wird mittels `os.system("adb shell monkey - p" + pacakge + "-v 1000")` ein automatisierter Durchlauf gestartet. Der Wert `-v 1000` steht hierbei für die Anzahl der auszuführenden Aktionen, welche zufällig auf dem Gerät ausgeführt werden. Nachdem die Aktionen ausgeführt wurden, wird zurück zum Ausgangspunkt navigiert. So ist die Wahrscheinlichkeit größer, dass die initiale Aktion in einen zuvor noch nicht getesteten Bereich der Anwendung führt und somit die Abdeckung erhöht. Ist die Anzahl der Durchläufe erreicht, wird der Benutzer abgemeldet und gegebenenfalls die Errors in einer Datei gespeichert. Die Umsetzung des Monkeyrunners befindet sich im Anhang unter dem Abschnitt A.2.

5.2.2 AndroidViewClient

Der Monkeyrunner liefert,bis auf die Angabe der Koordinaten, keine Lösung für das Interagieren mit einer Komponente. Dies stellt jedoch ein Problem dar, sollte sich das verwendete Layout grafisch dem Gerät anpassen. So sind die zu bedienenden Komponenten auf einem Handy und Tablet zwar dieselben, unterscheiden sich jedoch aufgrund der Orientierung und Größe des Displays in der dargestellten Position. Was dazu führt, dass der Befehl `device.touch(x, y, action)` auf dem jeweiligen Gerät eine unterschiedliche Aktion ausführen kann. Um die Schwierigkeit zu beheben, wurde der AndroidViewClient von Diego Torres Milano entwickelt. Hierbei handelt es sich ursprünglich um eine Erweiterung des Monkeyrunner, welche sich im Laufe der Zeit zu einem reinen Python-Tool entwickelte, das die Erstellung von Skripten zum Testen vereinfacht.[44] Eine der verwendeten Funktionen des Tools sind Interaktionen mit Views, welche unabhängig von der Bildschirmgröße, Auflösung oder Dichte sind. Das ist möglich, da die

Operationen durch View-Attribute anstelle von Koordinaten angegeben werden können. Möglich ist hierbei die Angabe des Textes, die ID oder der gegebene Inhalt. Im Unterschied zum Monkeyrunner wird im AndroidViewClient zusätzlich ViewClient mittels `vc = ViewClient(device, serialno)` zugewiesen. Anschließend wird die Methode `vc.dump()` ausgeführt, welche den Inhalt der Oberfläche zurück gibt. Mit `vc.traverse()` wird der Aufbau der Oberfläche durchlaufen und die Knoten ausgegeben. Dies kann die Zuweisung der Elemente vereinfachen. Ist das Element ausgewählt, kann mit `vc.findViewById("id/name").click()` ein einfacher Klick ausgeführt werden.[45] Die Änderungen, welche an dem ursprünglichen Code getätigt wurden, sind im Anhang unter dem Abschnitt A.3 zu finden.

6 Kontinuierliche Integration

Dieses Kapitel behandelt den theoretischen Aufbau eines Jenkins-Servers, welcher für die kontinuierliche Integration eines Android-Projektes verwendet werden kann. Hierzu wird die Verwendung der kontinuierlichen Integration, sowie das Erstellen eines Servers und den dazugehörigen Plugins beleuchtet.

6.1 Grundlagen

Bei der kontinuierlichen Integration handelt es sich um einen Begriff der Softwareentwicklung, welcher den Prozess des fortlaufenden Zusammenfügen der Software beschreibt. Um dies zu realisieren, wird ein Server zum Überwachen der Änderungen in einem Code-Repository eingesetzt. Tritt eine Änderung in dem Programm der Versionsverwaltung ein, kann eine Liste an Befehlen ausgeführt werden, um einen Build anzustoßen. So kann der Code nach einer Übergabe auf die Funktionsfähigkeit sowie gegebene Vorgaben geprüft werden. Der Nutzen besteht darin, dass eine unvoreingenommene Instanz die Kontrolle übernimmt. Sie ist dabei unabhängig von der Entwicklungsumgebung. Für die Kontrolle ist es notwendig, dass der Build die entsprechenden und ausführlichen Test-Suites beinhaltet. Um den Vorgang zu optimieren, wird vom Entwicklerteam täglich der Hauptzweig in den aktuellen Zweig übergeben und die dazugehörigen Tests sollten nicht länger als zehn Minuten benötigen. Durch das häufige Bauen und Testen der Anwendung kann die Softwarequalität gesteigert werden. Um diesen Effekt zu steigern, können in dem Schritt der Integration zusätzliche Softwaremetriken zur Messung der Softwarequalität angewandt werden. Da jedes Feature auf dem Code des Hauptzweiges basiert, sollte es sich bei diesem um ein Greenbuild handeln. Bei einem Greenbuild handelt es sich um das Ergebnis einer Codeänderung, welche alle am Build beteiligten Schritte durchlaufen hat und diese besteht. Hierdurch kann sichergestellt werden, dass die Codebasis nicht beschädigt ist. Sollte dies jedoch der Fall sein, hindert es das Team daran die folgenden Änderungen vertrauensvoll zu übergeben und erschwert die Suche nach einem Fehler.

Entwickelt das Team kleine Änderungen in inkrementen Schritten, bietet das den Vorteil, dass diese schnell in die Produktion übernommen werden können und Fehler leichter zu identifizieren sind. Ein Integrationsserver kann mit einem Tool wie Jenkins, TeamCity oder Bamboo selbst oder in einem System in der Cloud wie CloudBees oder Travis gehostet werden.[43]

6.2 Jenkins Server

Bei Jenkins handelt es sich um einen in Java geschriebenen und eigenständigen Open-Source-Automatisierungsserver und wurde initial von Kohsuke Kawaguchi entwickelt. Der Server dient zur Automatisierung aller Arten von Aufgaben im Zusammenhang mit dem Erzeugen, Testen, Ausliefern und Bereitstellen von Software.

6.2.1 Vorbereitung

Jenkins kann auf einem Gerät mit dem Betriebssystem Linux, macOS oder Windows installiert werden. Alternativ kann es in einer Umgebung von Docker oder Kubernetes realisiert werden. In Linux wird Jenkins mittels des Befehls `sudo apt-get install jenkins` installiert. Anschließend wird der Gruppe der Benutzernamen `jenkins` hinzugefügt, um dieser das Schreiben und Lesen innerhalb des Android-SDK-Ordners zu ermöglichen. Der Jenkins-Dienst wird nach dem Hochfahren des Gerätes automatisch gestartet und ist unter `http://localhost:8080` erreichbar. Die Adresse kann in den Einstellungen geändert werden. Nach der erfolgreichen Installation kann man unter “Select plugins to Install” die gewünschten Plugins installieren.[6] Für die zu testenden Android-Projekt im Zusammenhang mit Bitbucket können die Plugins JUnit, Bitbucket, Gradle und Android Emulator verwendet werden. Auf die jeweiligen Plugins wird im Folgenden genauer eingegangen. Nach der Installation der Plugins wird ein Admin-Benutzer angelegt und in den Einstellungen die Variable ANDROID_HOME und JAVA_HOME zugewiesen. In die Variablen ANDROID_HOME wird hierbei der Pfad des Android-SDK geschrieben um auf diese während des Builds zugreifen zu können. Sollte das Plugin kein gültiges SDK finden und die automatische Installation deaktiviert sein, wird der Build als “nicht gebaut” markiert und angehalten.

Abbildung 6.1: Jenkins Einstellungen

JUnit

Das JUnit-Plugin stellt einen Publisher zur Verfügung, welcher eine Web-UI zum Anzeigen von Testberichten und Verfolgen von Fehlern bietet. Zudem können XML-Testberichte, welche während eines Builds generiert und eingepflegt werden. Weiter können die historischen Testergebnisse grafisch dargestellt werden. Der JUnit-Publisher kann in der Ebene der Jobs konfiguriert werden, indem eine Aktion zum Veröffentlichen von JUnit-Testbericht nach einem Build hinzugefügt wird. Hierbei kann die Prüfung der Veröffentlichung übersprungen, der Namen der Überprüfung angepasst und Standardausgaben oder Fehler einem Test-Suite in den Testergebnissen beibehalten werden.[8]

Bitbucket

Mit dem Bitbucket-Plugin kann ein Bitbucket-Server mit Jenkins verbunden werden. So ist es möglich, automatisch ein Webhook in Bitbucket zu erstellen, um Builds auszulösen und von diesem zu klonen. Hierfür kann das Plugin die Anmeldeinformationen sicher verwahren. Empfohlen wird ein Benutzer mit Leserechten zu erstellen und diesen zu verwenden. Zudem können detaillierte Build-Informationen wie die Zusammenfassung oder Dauer der Tests angezeigt werden. Ist es gewünscht, kann der Build-Status automatisch an den Bitbucket Server gesendet werden. Um das Plugin verwenden zu können, wird die Version Jenkins 2.204.4 und Bitbucket 7.4 oder höher benötigt. Um die Informationen des Bitbucket-Servers hinterlegen zu können, wird zunächst in Bitbucket unter den persönlichen Zugangstoken ein neuer Token erstellt. Der erzeugte Token wird anschließend, wie der Name und die URL des Servers, in den Einstellungen des Jenkins-Servers unter

dem Punkt "Bitbucket-Server" angegeben. Zudem kann noch ein Berechtigungsnachweis hinterlegt werden, welcher das Zuweisen in einem Job erleichtert.[5]

Gradle

Das Gradle Plugin wird, wie Ant oder Maven, für die automatische Installation verwendet. Außerdem stellt es einen weiteren Build-Schritt bereit, um eine Aufgabe in Gradle auszuführen. Zudem ermöglicht es Scans der Builds in beliebigen Protokollen für Maven- und Gradle-Builds zu erkennen und diese in der Oberfläche von Jenkins anzuzeigen.[7]

Android Emulator

Ein essentielles Plugin für das Testen von Android-Anwendungen ist der Android Emulator. Mit diesem kann beim Erstellen eines Jobs ein Emulator während des Builds hinterlegt werden. Hierbei kann ein Android Virtual Devices oder ein in Jenkins erstellter Emulator verwendet werden. Wird ein Emulator in Jenkins erstellt, können zum Beispiel die OS-Version, Rasterdichte, Auflösung und Sprache angegeben werden. Alternativ dazu kann ein Schnappschuss eines Emulators verwendet werden.[4]

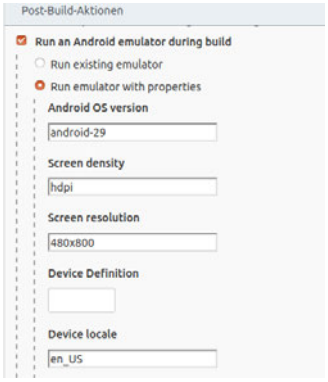


Abbildung 6.2: Jenkins Android-Emulator

Des Weiteren kann durch das Plugin unter dem Punkt "Buildverfahren" ein Monkeyrunner ausgeführt werden. Um diesen zu verwenden, wird die ID des Paketes sowie die Anzahl der auszuführenden Aktionen angegeben. Zudem kann eine Verzögerung zwischen den Aktionen hinzugefügt werden. Alternativ dazu kann ein Befehl in der Shell ausgeführt werden, welcher das Skript des Monkeyrunners aus Abschnitt 5.2 beinhaltet.[4]

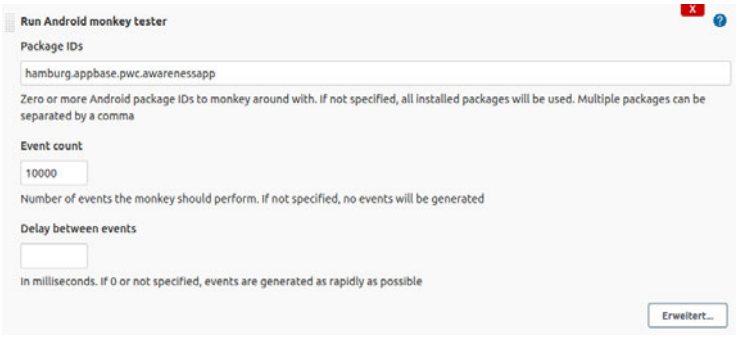


Abbildung 6.3: Jenkins Monkeyrunner

Sollte `connectedDebugAndroidTest` in dem Buildverfahren angegeben werden, führt dies zum Ausführen der Android-Tests auf einem tatsächlichen Android-Gerät, welches mit dem Server verbunden ist oder wie oben gezeigt erstellt wurde.[4]

6.2.2 Erstellen eines Jobs

Um das Projekt zu testen wird zunächst ein neuer Freestyle- oder Pipeline-Job erstellt. Bei einem Freestyle-Job handelt es sich um den meistgenutzte Job in Jenkins. Hiermit kann ein Projekt gebaut werden, wobei ein beliebiges Programm zur Versionsverwaltung mit jedem Build-Verfahren kombiniert werden kann. Zudem können Plugins wie der Android Emulator, Bitbucket oder Gradle verwendet werden. Eine Pipeline unterscheidet sich im im Vergleich zum Freestyle-Job darin, dass der Job auf dem Master von Jenkins ausgeführt wird. Hierbei wird ein leichtgewichtiger Executor verwendet, welcher wenige Ressourcen benötigt und die Pipeline in atomare Befehle übersetzt, ausführt und an die Agenten verteilt. So kann der Job in mehrere Phasen unterteilt werden. Nach einer Phase kann eine Verifizierung hinzugefügt werden, bevor fortgefahren wird. Dies ist in einem Freestyle-Job nicht möglich. Zudem besteht die Möglichkeit mehrere Tests oder Phasen parallel durchzuführen.[6]

Um den Job zu erstellen, wird zunächst der Name dessen angegeben und in diesem Fall ein Freestyle-Job ausgewählt. Anschließend wird unter dem Punkt “General” die Projekt-URL und dessen Name festgelegt. Zusätzlich kann eine Beschreibung eingetragen werden. Danach wird im Source-Code-Management die Quelle des zu testenden Codes hinterlegt. Hierbei muss für Bitbucket die Serverinstanz und der Projektname angegeben werden. Ist der Server bereits hinterlegt, kann dieser angegeben werden. Ist dies nicht der Fall, muss ein Berechtigungsnachweis und optional der SSH-Berechtigungsnachweis

hinzugefügt werden. Zusätzlich zu dem Repository kann der Zweig angegeben werden, aus welchem das Projekt geklont werden soll.

Um den Build auszulösen, kann zwischen mehreren Optionen gewählt werde. So ist es möglich, den Build durch ein Skript von außerhalb, zeitgesteuert oder durch einen Anstoß zu starten. Unter dem Punkt Build-Auslöser kann `Bitbucket Server trigger build after push` aktiviert werden. Ist der Bitbucket-Server, wie eingangs beschrieben, eingerichtet, wird nach einer Übergabe des Codes an den Server der Build-Prozess ausgelöst. Alternativ kann der Build-Prozess manuell gestartet werden. Anschließend kann in der Buildumgebung eine Zeit bis zum Abbruch des Builds angegeben werden. Zudem kann die Anweisung gegeben werden, dass bei dem Build ein Android-Emulator verwendet werden soll. Dieser kann beim Buildverfahren zugewiesen oder erstellt werden. Des weiteren können Build-Schritte hinzugefügt werden. Hier kann das Gradle-Skript aufgerufen werden und führt die angegebenen Aufgaben durch. Mittels `clean` können alle früheren Builds gelöscht werden. Hierdurch kann sichergestellt werden, dass nichts zwischengespeichert und ein sauberer Build verwendet wird. Mit `assembleDebug` wird die Debug.apk erzeugt und `test` führt die JUnit-Tests in allen Modulen aus.[7]

Unter dem letzten Punkt “Post-Build-Aktionen” kann ausgewählt werden, was nach dem Build ausgeführt wird. So kann beispielhaft das Testergebnis von JUnit oder das Javadoc veröffentlicht, weitere Projekte gebaut oder die Ergebnisse mittels einer E-Mail versendet werden. Zudem kann der Arbeitsbereich gelöscht und der Status in Git oder Bitbucket für den jeweiligen Commit aktualisiert werden. Nach diesem Schritt kann der Job gespeichert und verwendet werden. Wird anschließend der Build angestoßen, kann der Prozess in der Konsolenausgabe nachvollzogen werden. Nachdem der Emulator gestartet ist, wird die Anwendung gebaut und installiert. Sollte es zu einem Fehler in den Tests kommen, können diese anschließend ausgelesen oder in einem Dokument gespeichert werden. Ist die Ursache des Fehlers gefunden und behoben, wird der Prozess durch eine erneute Übergabe des Codes an das Programm zur Versionsverwaltung erneut gestartet.

7 Fazit und Ausblick

7.1 Fazit

Um die Qualität der Software auf lange Sicht anzuheben, sollte ein Unternehmen sich für eine Teststrategie entscheiden. Diese Entscheidung sollte vor dem Start der Entwicklung getroffen werden. Handelt es sich bei der Strategie um die Verwendung von automatisierten Tests, kann bei einer längeren Entwicklung der Software Geld gespart werden, da bereits geschriebene Tests mehrmals verwendet werden können. Die testgetriebene Entwicklung hat zudem den Vorteil, dass das Schreiben der Tests nicht aufgeschoben werden kann. Hierdurch steht das Entwicklerteam vor einer Veröffentlichung nicht vor dem Problem, dass die Software noch schnell und gegebenenfalls händisch getestet werden muss.

Das Ziel von automatisierten Tests sollte sein, dass diese in einer möglichst realistischen und kostengünstigen Umgebung ausgeführt werden. Hierzu können zum größten Teil Emulatoren verwendet und in der finalen Testphase mit physischen Geräten ergänzt werden. Hierdurch können mit geringen Kosten viele Spezifikationen, aber auch eine reale Umgebung getestet werden.

In den Modul- und Integrationstests ist es notwendig Mockups zu verwenden, um diese Unabhängig von anderen Modulen oder Systemen ausführen zu können. In einem Projekt, welches in Kotlin geschrieben wird, empfiehlt sich die verwendung von MockK. Das Schreiben der Modul- und Integrationstests ist im Vergleich zu den Tests der Benutzeroberfläche sehr zeitaufwändig. Jedoch können diese das Vertrauen in den geschrieben Code erhöhen. Zudem kann davon ausgegangen werden, dass die Funktionen, Module und Komponenten nach einem Update, wie zum Beispiel der verwendeten Bibliotheken, wie erwartet funktionieren. Das Durchführen eines Integrationstests in Verbindung mit dem MVVM-Pattern und der intern verwendeten Bibliothek konnte nicht ohne Probleme durchgeführt werden. Daher ist es eine Überlegung wert, die Bibliothek in Java zu

kompilieren und auszuliefern, um dies in zukünftigen Projekten zu beheben. Um ein Android-Projekt mit den MVVM-Muster zukünftig leichter testen zu können, sollte die Struktur zudem modularer gestaltet werden. So kann für größere Teile des Codes Modultests geschrieben und ohne Emulator ausgeführt werden, auch wenn die Bibliothek nicht in Java übersetzt wird.

Um die Benutzeroberfläche zu testen, eignete sich das Aufnehmen der User Story mittels des Espresso Test Recorders. Die Aufnahme kann auch von Mitarbeitern ohne tiefere Kenntnisse in Kotlin oder Java durchgeführt werden. Anschließend können diese erweitert und gegebenenfalls angepasst werden. Die Kombination aus den Tests der User Storys und einem automatischen Klicker ermöglicht ein umfangreiches Testen der Anwendung. Um eine Anwendung zu testen, kann daher eine Kombination aus Modul- und Integrationstest sowie Tests der Benutzeroberfläche umgesetzt werden. Zusätzlich zu den automatisierten Tests sollte der Prozess zusätzlich durch manuelles Testen erweitert werden.

Um den Prozess weithin zu automatisieren, ist der Einsatz eines Servers zur kontinuierlichen Integration geeignet. Führt dieser die Testroutine automatisiert und nach der Übergabe des Codes in das Programm der Versionsverwaltung durch, kann dies das Vertrauen in den Code und dessen Qualität erhöhen. Abhängig von der Qualität der Tests kann zudem davon ausgegangen werden, dass es durch Änderungen, Updates oder Erweiterungen zu keinem Fehlverhalten in der Anwendung kommt.

7.2 Ausblick

Um die Qualität der Anwendung langfristig zu steigern, könnten die an den Modulen und Komponenten gezeigten Methoden auf die gesamte Anwendung erweitert werden. Wird die Anwendung um die entsprechenden Tests erweitert, kann der Entwicklungsprozess um den Einsatz eines Servers der kontinuierlichen Integration erweitert werden. Für diesen Server kann in zukünftigen Schritten zudem eine Gerätefarm gebaut werden. Diese soll eine vielzahl an Emulatoren mit unterschiedlichsten Konfigurationen beinhalten. Zudem könnte ein System der Cloud hinzugefügt werden, welches physische Geräte eingebunden hat und diese verwaltet. So soll es möglich sein die automatisierten Tests auf beliebig vielen oder zufällig gewählten Geräten ausführen zu lassen. Alternativ kann auch über die Nutzen einer bereits bestehenden Farm, wie die AWS-Gerätefarm, nachgedacht werden. Des weiteren kann der Server der kontinuierlichen Integration um eine Plattform wie

SonarQube erweitert werden. SonarQube ist eine Werkzeug zur kontinuierlichen Überprüfung der Qualität und Sicherheit der Codebasis.[49]

Wird weder eine Gerätefarm gemietet noch aufgebaut, können die Tests um eine erweiterte Simulation der Umgebung für Emulatoren ergänzt werden. Hierbei kann zum Drehen des Gerätes das Verhalten um einen variablen Akku oder Ladestand erweitert werden. Zudem kann die Signalstärke des Netzwerks zufällig geändert und zwischen WLAN und den Mobilen Daten gewechselt werden.

Der Server der kontinuierlichen Integration kann zudem um die Komponente der kontinuierlichen Auslieferung erweitert werden. Hierbei handelt es sich um eine Sammlung an Techniken, Prozessen und Werkzeugen, welche den Prozess der kontinuierlichen Auslieferung verbessern und automatisieren können. Hierdurch können die von einem Server der kontinuierlichen Integration erzeugten Versionen einer Software automatisiert auf Entwicklungs-, Test-, Integrations- und Produktivumgebung eingespielt werden.[2]

Das Erstellen der Tests einer User Stories mittels des Espresso Test Recorders kann zukünftig von Personal durchgeführt werden, welches nicht direkt an der Entwicklung der Anwendung beteiligt ist. So kann zukünftig Zeit des Entwicklerteams eingespart und eine unabhängige Kontrollinstanz hinzugefügt werden. Durch das Umsetzen der Testautomatisierung soll langfristig die Qualität und Zuverlässigkeit der ausgelieferten Software gesteigert und eine Kultur des Testens eingeführt werden.

Literaturverzeichnis

[1] A. Contan, L. M.: *Test automation pyramid from theory to practice.* 2012. – URL https://ieeexplore.ieee.org/abstract/document/8402699. – Zugriffsdatum: 2020-09-27

[2] A. Poth, X. L.: *Systems, Software and Services Process Improvement How to Deliver Faster with CI/CD Integrated Testing Services?* Springer, 2018. – URL https://link.springer.com/book/10.1007/978-3-319-97925-0. – ISBN 9783319979250

[3] Bezsmolna, Victoria: *Real Devices vs. Emulators for Reliable Mobile App Testing.* – URL https://bitbar.com/blog/real-devices-vs-emulators-for-reliable-mobile-app-testing/. – Zugriffsdatum: 2020-12-13

[4] CloudBees: *Jenkins Android Emulator.* – URL https://plugins.jenkins.io/android-emulator/. – Zugriffsdatum: 2021-04-10

[5] CloudBees: *Jenkins Bitbucket Server Integration.* – URL https://plugins.jenkins.io/atlassian-bitbucket-server-integration/. – Zugriffsdatum: 2021-04-09

[6] CloudBees: *Jenkins Dokumentation.* – URL https://www.jenkins.io/doc/. – Zugriffsdatum: 2021-04-12

[7] CloudBees: *Jenkins Gradle.* – URL https://plugins.jenkins.io/gradle/. – Zugriffsdatum: 2021-04-10

[8] CloudBees: *Jenkins JUnit.* – URL https://plugins.jenkins.io/junit/. – Zugriffsdatum: 2021-04-09

[9] Dictonary, Cambridge: *Mockup Definition.* 2020. – URL https://dictionary.cambridge.org/dictionary/english/mock-up. – Zugriffsdatum: 2020-12-01

[10] Dijkstra, Prof.dr. Edsger W.: *NOTES ON STRUCTERD PROGRAMMING*. 1970. – URL https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF. – Zugriffsdatum: 2020-09-24

[11] Flower, Martin: *UnitTest*. – URL https://martinfowler.com/bliki/UnitTest.html. – Zugriffsdatum: 2020-12-29

[12] Flower, Martin: *Test Driven Development*. 2005. – URL https://martinfowler.com/bliki/TestDrivenDevelopment.html. – Zugriffsdatum: 2020-10-30

[13] Flower, Martin: *Test Double*. 2006. – URL https://martinfowler.com/bliki/TestDouble.html. – Zugriffsdatum: 2020-12-01

[14] Flower, Martin: *Self Testing Code*. 2014. – URL https://martinfowler.com/bliki/SelfTestingCode.html. – Zugriffsdatum: 2020-11-03

[15] Franz, Klaus: *Planung der Teststufen*. 2014. – URL https://link.springer.com/chapter/10.1007/978-3-662-44028-5_16. – Zugriffsdatum: 2020-09-24

[16] Google: *Firebase Test Lab*. – URL https://firebase.google.com/docs/test-lab. – Zugriffsdatum: 2020-12-13

[17] Google, JetBrains: *Android Profiler*. – URL https://developer.android.com/studio/profile/android-profiler. – Zugriffsdatum: 2020-12-13

[18] Google, JetBrains: *Build and run your app*. – URL https://developer.android.com/studio/run. – Zugriffsdatum: 2021-03-25

[19] Google, JetBrains: *Build local unit tests*. – URL https://developer.android.com/training/testing/unit-testing/local-unit-tests. – Zugriffsdatum: 2021-01-09

[20] Google, JetBrains: *Create UI tests with Espresso Test Recorder*. – URL https://developer.android.com/studio/test/espresso-test-recorder. – Zugriffsdatum: 2021-04-12

[21] Google, JetBrains: *Espresso*. – URL https://developer.android.com/training/testing/espresso/basics. – Zugriffsdatum: 2020-01-09

[22] Google, JetBrains: *monkeyrunner*. – URL https://developer.android.com/studio/test/monkeyrunner. – Zugriffsdatum: 2021-03-20

[23] Google, JetBrains: *ViewModel Overview*. – URL https://developer.android.com/topic/libraries/architecture/viewmodel. – Zugriffsdatum: 2021-01-09

[24] Google, JetBrains: *Add build dependencies*. 2020. – URL https://developer.android.com/studio/build/dependencies. – Zugriffsdatum: 2020-10-21

[25] Google, JetBrains: *Fundamentals of Testing*. 2020. – URL https://developer.android.com/training/testing/fundamentals. – Zugriffsdatum: 2020-10-24

[26] Groups, Google: *Dexing*. – URL https://docs.elementscompiler.com/Platforms/Android/BuildPhases/Dexing/. – Zugriffsdatum: 2021-01-02

[27] Groups, Google: *Robolectric*. – URL http://robolectric.org/. – Zugriffsdatum: 2020-11-15

[28] Guru: *Real Device Vs Simulator Vs Emulator Testing: Key Differences*. 2020. – URL https://www.guru99.com/real-device-vs-emulator-testing-ultimate-showdown.html. – Zugriffsdatum: 2020-12-12

[29] H. Kaur Brar, P. Jai K.: *Differentiating Integration Testing and unit testing*. – URL https://ieeexplore.ieee.org/abstract/document/7100358. – Zugriffsdatum: 2021-01-15

[30] H. Saiedian, D. J.: *Test-driven development concepts, taxonomy, and future direction*. 2005. – URL https://ieeexplore.ieee.org/document/1510569. – Zugriffsdatum: 2020-10-30

[31] JetBrains: *Object Expressions and Declarations*. – URL https://kotlinlang.org/docs/reference/object-declarations.html. – Zugriffsdatum: 2021-01-09

[32] JUnit-Team: *Class Assert*. – URL https://junit.org/junit4/javadoc/latest/org/junit/Assert.html. – Zugriffsdatum: 2021-01-09

[33] JUnit-Team: *junit4 - Assertion*. – URL https://github.com/junit-team/junit4/wiki/Assertions. – Zugriffsdatum: 2021-01-09

[34] Kent Beck, Cynthia A.: *Extreme Programming Explained Embrace Change*. Addison-Wesley Professional, 2004. – URL https://books.google.de/books?id=G8EL4H4vf7UC&redir_esc=y. – ISBN 0201616416

[35] Kneuper, Ralf: *Die geschichtliche Entwicklung des V-Modells*. 2018. – URL https://www.researchgate.net/publication/328955806_Die_geschichtliche_Entwicklung_des_V-Modells. – Zugriffsdatum: 2020-10-24

[36] Lipski, Michal: *Test Doubles - Fakes, Mocks and Stubs*. 2017. – URL https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da. – Zugriffsdatum: 2020-12-06

[37] M. Ellims, D I.: *The Economics of Unit Testing*. – URL https://link.springer.com/article/10.1007/s10664-006-5964-9. – Zugriffsdatum: 2020-01-09

[38] M. Jarke, J. Mylopoulos & c.: *Advanced Information Systems Engineering*. O'Reilly Media, 2014. – URL https://link.springer.com/book/10.1007/978-3-319-07881-6. – ISBN 9783662175378

[39] M. Linares-Vasquez, K. Moran & c.: *How do Developers Test Android Applications?*. – URL https://ieeexplore.ieee.org/abstract/document/8094467. – Zugriffsdatum: 2021-03-04

[40] M. LinaresVasquez, C. BernalCardenas K. Moran D. P.: *How do Developers Test Android Applications?*. – URL https://ieeexplore.ieee.org/document/8094467. – Zugriffsdatum: 2020-01-15

[41] M. Pol, A. S.: *Management und Optimierung des Testprozesses Praktischer Leitfaden für erfolgreiches Software-Testen mit TPI und TMap*. Random House Incorporated, 2002. – URL https://books.google.de/books?id=RDy4ygAACAAJ. – ISBN 9780345397041

[42] M. Torchiano, R. C.: *Automated mobile UI test fragility An exploratory assessment study on Android*. – URL https://dl.acm.org/doi/abs/10.1145/2945404.2945406. – Zugriffsdatum: 2021-04-10

[43] Meyer, Mathias: *Continuous Integration and Its Tools*. – URL https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6802994. – Zugriffsdatum: 2021-04-09

[44] Milano, Diego T.: *AndroidViewClient*. – URL https://github.com/dtmilano/AndroidViewClient/wiki. – Zugriffsdatum: 2021-03-25

[45] Milano, Diego T.: *AndroidViewClient Lib*. – URL https://dtmilano.github.io/AndroidViewClient/. – Zugriffsdatum: 2021-03-25

[46] Pylypenko, Oleksiy: *MockK Github*. 2020. – URL https://github.com/mockk/mockk. – Zugriffsdatum: 2020-11-12

[47] Sanket: *The exponential cost of fixing bugs*. 2019. – URL https://deepsource.io/blog/exponential-cost-of-fixing-bugs/. – Zugriffsdatum: 2020-12-21

[48] Slany, Hirsch Schindler Müller S.: *An Approach to Test Classification in Big Android Applications*. – URL https://ieeexplore.ieee.org/document/8859509. – Zugriffsdatum: 2020-01-15

[49] SonarSource: *About SonarQube*. – URL https://www.sonarqube.org/about/. – Zugriffsdatum: 2021-05-01

[50] White, Kalei: *Increase Efficiency and Productivity with Mobile App Test Automation*. 2016. – URL https://abstracta.us/blog/mobile-testing/increase-efficiency-and-productivity-with-mobile-app-test-automation/. – Zugriffsdatum: 2020-11-13

A Anhang

A.1 Integrationstests LoginFragment

```
@RunWith(AndroidJUnit4::class)
@MediumTest
class FragmentLoginTest {

    @MockK
    private lateinit var scenario: FragmentScenario<*>
    private lateinit var instrumentationContext: Context
    private lateinit var loginViewModel: LoginViewModel
    private val mockTheme = mockk<Theme> {
        every { _id } returns 2
        every { mainColor } returns "13262bFF"
        every { secondColor } returns "f4f4f4FF"
        every { accentColor } returns "ef8100FF"
        every { backgroundColor } returns "005873FF"
        every { mainTextColor } returns "f4f4f4FF"
        every { secondTextColor } returns "3a3a3aFF"
        every { mainInputBackgroundColor } returns "f4f4f440"
        every { secondInputBackgroundColor } returns "f4f4f4BF"
        every { mainInputPlaceholderColor } returns "f4f4f480"
        every { secondInputPlaceholderColor } returns "3a3a3a80"
        every { popupBackgroundColor } returns "f4f4f4FF"
        every { postedAtColor } returns "f4f4f4FF"
        every { errorColor } returns "f4f4f4FF"
        every { successColor } returns "8bc34aFF"
        every { onErrorColor } returns "c62828FF"
        every { textSize } returns "m"
        every { logo } returns null
        every { icon } returns null
    }

    @Before fun setup() {
        MockKAnnotations.init(this)
        instrumentationContext = InstrumentationRegistry.
                getInstrumentation().targetContext

        mockkClass(HStore::class)
        loginViewModel = spyk(LoginViewModel())
```

```
        loginViewModel.theme.postValue(mockTheme)
        loginViewModel.showRegistration.postValue(true)
        loginViewModel.companyName.postValue("Appbase GmbH")
        scenario = lunchLoginFragment(loginViewModel)
    }

    @Test
    fun checkLoginFragmentUIElements() {
        //UI element Title
        val title = onView(withId(R.id.textView))
        title.perform(scrollTo())
        title.check(matches(isDisplayed()))
        title.check(matches(withText("Log in")))
        title.check(withTextViewColorAssertions(textColor = mockTheme.mainText))
        title.check(withTextViewSizeAssertions(textSize = mockTheme.fontSizeTitle))
        title.check(withTextViewTypefaceAssertions(typeface = Store.typeFaceTitle))
        title.check(matches(withTextViewSizeMatcher(12)))

        //UI element EmailInput
        val emailInput = onView(withId(R.id.email_input))
        emailInput.perform(scrollTo())
        emailInput.check(matches(isDisplayed()))
        InputField.checkBasics(parentId = R.id.email_input, title = "Email",
                icon = ICON.NONE)
        InputField.checkColor(
            parentId = R.id.email_input, titleColor = mockTheme.mainText,
            contentColor = mockTheme.mainText, backgroundColor = mockTheme.mainInput
        )
        InputField.checkInputType(
            parentId = R.id.email_input,
            type = InputType.TYPE_TEXT_VARIATION_EMAIL_ADDRESS
        )
        InputField.checkTextSize(
            parentId = R.id.email_input,
            textSize = mockTheme.fontSizeContent,
            titleTextSize = instrumentationContext.resources
                .getDimension(R.dimen.font_input_title).toInt()
        )

        //UI element PasswordInput
        val passwordInput = onView(withId(R.id.password_input))
        passwordInput.perform(scrollTo())
        passwordInput.check(matches(isDisplayed()))
        InputField.checkBasics(parentId = R.id.password_input,
                title = "Password", icon = ICON.PASSWORD)
        InputField.checkColor(
            parentId = R.id.password_input, titleColor = mockTheme.mainText,
            contentColor = mockTheme.mainText, backgroundColor
                = mockTheme.mainInput
        )
        InputField.checkInputType(
```

```
        parentId = R.id.password_input,
        type = InputType.TYPE_CLASS_TEXT +
            InputType.TYPE_TEXT_VARIATION_PASSWORD
    )
    InputField.checkTextSize(
        parentId = R.id.password_input,
        textSize = mockTheme.fontSizeContent,
        titleTextSize = instrumentationContext.resources
            .getDimension(R.dimen.font_input_title).toInt()
    )

    //UI element ForgotPassword
    val passwordForgot = onView(withId(R.id.forgot_password))
    passwordForgot.perform(scrollTo())
    passwordForgot.check(matches(isDisplayed()))
    passwordForgot.check(matches(withText("Forgot␣Password?")))
    passwordForgot.check(withTextViewColorAssertions(textColor =
            mockTheme.accent))
    passwordForgot.check(withTextViewTypefaceAssertions(Store.typeFace))

    //UI element LoginBtn
    val loginBtn = onView(withId(R.id.include))
    loginBtn.perform(scrollTo())
    ButtonStyled.checkBasic(R.id.include, "Log␣in")
    ButtonStyled.checkColors(
        parentId = R.id.include,
        textColor = mockTheme.secondary,
        backgroundColor = mockTheme.accent
    )
    ButtonStyled.checkTypeface(parentId = R.id.include,
            typeface = Store.typeFaceTitle)
    ButtonStyled.checkTextSize(parentId = R.id.include,
            textSize = mockTheme.fontSizeButton)

    //UI element PrivacyPolice
    val privacyPolice = onView(withId(R.id.privacy_police))
    privacyPolice.perform(scrollTo())
    privacyPolice.check(matches(isDisplayed()))
    privacyPolice.check(matches(withText("Read␣our␣privacy␣policy")))
    privacyPolice.check(withTextViewColorAssertions(textColor =
            mockTheme.accent))
    privacyPolice.check(
            withTextViewSizeAssertions(textSize =
                    mockTheme.fontSizeContent))
    privacyPolice.check(withTextViewTypefaceAssertions(Store.typeFace))

    //UI element SignUpTitle
    val signUpTitle = onView(withId(R.id.sign_up_title))
    signUpTitle.perform(scrollTo())
    signUpTitle.check(matches(isDisplayed()))
    signUpTitle.check(matches(withText("Don't␣have␣an␣account␣yet?")))
```

```
        signUpTitle.check(withTextViewColorAssertions(
                textColor = mockTheme.mainText))
        signUpTitle.check(withTextViewSizeAssertions(
                textSize = mockTheme.fontSizeContent))
        signUpTitle.check(withTextViewTypefaceAssertions(Store.typeFace))

        //UI element SignUpBtn
        val signUpBtn = onView(withId(R.id.include_sign_up))
        signUpBtn.perform(scrollTo())
        ButtonStyled.checkBasic(R.id.include_sign_up, "Sign up")
        ButtonStyled.checkColors(
            parentId = R.id.include_sign_up,
            textColor = mockTheme.accent,
            backgroundColor = mockTheme.secondary
        )
        ButtonStyled.checkTypeface(parentId = R.id.include_sign_up,
                typeface = Store.typeFaceTitle)
        ButtonStyled.checkTextSize(parentId = R.id.include_sign_up,
                textSize = mockTheme.fontSizeButton)

        //UI element LoginError
        val loginError = onView(withId(R.id.login_error))
        loginError.check(matches(Matchers.not(isDisplayed())))
    }

    @Test
    fun loginDifferentInput() {
        every { Store.authenticator.default.login("testmail@appbase.hamburg",
                "password123", null) } returns
                        DeferredObject<String, Exception, Void>()
                                .reject(IllegalStateException(
                                        "Deferred object rejected"))

        val loginError = onView(withId(R.id.login_error))
        val loginBtn = onView(withId(R.id.include))

        //password visibility button
        InputField.setContent(parentId = R.id.password_input,
                input = "password123")
        InputField.checkIconState(parentId = R.id.password_input,
                true, ICON.PASSWORD)
        InputField.checkInputType(parentId = R.id.password_input,
                InputType.TYPE_CLASS_TEXT + InputType.TYPE_TEXT_VARIATION_PASSWORD)
        InputField.performIconClick(parentId = R.id.password_input, ICON.PASSWORD)
        InputField.checkInputType(parentId = R.id.password_input,
                InputType.TYPE_CLASS_TEXT +
                        InputType.TYPE_TEXT_VARIATION_VISIBLE_PASSWORD)

        //login without data
        loginBtn.perform(scrollTo())
        ButtonStyled.performClick(parentId = R.id.include)
```

```
        loginError.perform(scrollTo())
        loginError.check(matches(isDisplayed()))
        loginError.check(matches(withText("Please enter your email and password.")))

        InputField.checkIconState(parentId = R.id.email_input, true, ICON.DELETE)
        InputField.checkIconState(parentId = R.id.password_input, true, ICON.DELETE)

        //login with wrong data
        InputField.setContent(parentId = R.id.email_input,
                input = "testmail@appbase.hamburg")
        InputField.setContent(parentId = R.id.password_input,
                input = "password123")

        ButtonStyled.performClick(parentId = R.id.include)

        loginError.check(matches(isDisplayed()))
        loginError.check(matches(withText(
                "We're sorry, but the combination doesn't seem to exist."+
                        " Please check your email and/or your password.")))

        //clear fields
        InputField.performIconClick(parentId = R.id.email_input, ICON.DELETE)
        InputField.performIconClick(parentId = R.id.password_input, ICON.DELETE)

        InputField.checkContent(parentId = R.id.email_input, "")
        InputField.checkContent(parentId = R.id.password_input, "")
    }

    @After
    fun cleanUp() {
        unmockkAll()
    }
}
```

Quellcode A.1: Integrationstests LoginFragment

A.2 Monkeyrunner

```
# Imports the monkeyrunner modules used by this program
from com.android.monkeyrunner import MonkeyRunner, MonkeyDevice
from java.util.logging import Level, Logger, StreamHandler, SimpleFormatter
from java.io import ByteArrayOutputStream
import os
import time


# Connects to the current device, returning a MonkeyDevice object
device = MonkeyRunner.waitForConnection()
```

```
# Installs the Android package. Notice that this method returns a boolean,
# so you can test to see if the installation worked.
device.installPackage('/Users/simonborho/Development/' +
        'MonkeyTester/pwc-awarenessapp-v1.0.8.apk')

# sets a variable with the package's internal name
package = 'hamburg.appbase.pwc.awarenessapp'

# sets a variable with the name of an Activity in the package
activity = 'hamburg.appbase.pwc.awarenessapp.MainActivity'
# sets the name of the component to start
runComponent = package + '/' + activity
errors = ByteArrayOutputStream(100)
logger = Logger.getLogger('com.android.chimpchat.adb.AdbChimpDevice')
logger.addHandler(StreamHandler(errors, SimpleFormatter()))

# Runs the component
device.startActivity(component=runComponent)

# - LOG IN - #
device.startActivity(component=runComponent)

# click and write into email field
device.touch(215, 680, MonkeyDevice.DOWN_AND_UP)
device.type('testuser@appbase.hamburg')

# click and write into password field
device.touch(215, 920, MonkeyDevice.DOWN_AND_UP)
device.type('Test*123tst')

#click login
device.touch(520, 1200, MonkeyDevice.DOWN_AND_UP)

#enter verification code
device.touch(200, 1100, MonkeyDevice.DOWN_AND_UP)
device.type('C3168G')
device.touch(200, 1100, MonkeyDevice.DOWN_AND_UP)

# - Random clicker - #
for x in range(2):
        for x in range(3):
                device.touch(70, 125, MonkeyDevice.DOWN_AND_UP)
                time.sleep(1)
        os.system('adb shell monkey -p' +
                'hamburg.appbase.pwc.awarenessapp -v 1000')

# - LOG OUT - #
device.startActivity(component=runComponent)
device.touch(1030, 180, MonkeyDevice.DOWN_AND_UP)
device.touch(125, 1920, MonkeyDevice.DOWN_AND_UP)
```

```
try(OutputStream outputStream = new FileOutputStream('errorLog')) {
    byteArrayOutputStream.writeTo(outputStream);
}
MonkeyRunner.alert('Test beendet', 'Ende', 'OK')
```

Quellcode A.2: Monkeyrunner

A.3

Monkeyrunner mit AndroidViewClient

```
# Imports the monkeyrunner modules used by this program
from com.android.monkeyrunner import MonkeyRunner, MonkeyDevice
from java.util.logging import Level, Logger, StreamHandler, SimpleFormatter
from java.io import ByteArrayOutputStream
import os
import time
import re
import sys


try:
    sys.path.insert(0, os.path.join(os.environ['/User/simonborho/Development' +
        '/AndroidViewClient'], 'src'))
except:
    pass

from com.dtmilano.android.viewclient import ViewClient

device = MonkeyRunner.waitForConnection()
device.installPackage('/Users/simonborho/Development/MonkeyTester/'+
        'pwc-awarenessapp-v1.0.8.apk')
package = 'hamburg.appbase.pwc.awarenessapp'
activity = 'hamburg.appbase.pwc.awarenessapp.MainActivity'
runComponent = package + '/' + activity
errors = ByteArrayOutputStream(100)
logger = Logger.getLogger('com.android.chimpchat.adb.AdbChimpDevice')
logger.addHandler(StreamHandler(errors, SimpleFormatter()))
device.startActivity(component=runComponent)

# - LOG IN - #
device.startActivity(component=runComponent)
device.touch(By.id('email_input'), MonkeyDevice.DOWN_AND_UP)
device.type('testuser@appbase.hamburg')
click and write into password field
device.touch(By.id('password_input'), MonkeyDevice.DOWN_AND_UP)
device.type('Test*123tst')
device.touch(By.id('include'), MonkeyDevice.DOWN_AND_UP)
device.touch(By.id('auth_input'), MonkeyDevice.DOWN_AND_UP)
```

```
device.type('C3168G')
device.touch(By.id('auth_button'), MonkeyDevice.DOWN_AND_UP)

# - Random clicker - #
for x in range(2):
        for x in range(3):
                device.touch(By.id('home'), MonkeyDevice.DOWN_AND_UP)
        os.system('adb shell monkey -p ' +
                'hamburg.appbase.pwc.awarenessapp -v 1000')

# - LOG OUT - #
device.startActivity(component=runComponent)
device.touch(By.id('imageButtonId'), MonkeyDevice.DOWN_AND_UP)
device.touch(By.id('logout'), MonkeyDevice.DOWN_AND_UP)
try(OutputStream outputStream = new FileOutputStream('errorLog')) {
    byteArrayOutputStream.writeTo(outputStream);
}
MonkeyRunner.alert('Test beendet', 'Ende', 'OK')
```

Quellcode A.3: Monkeyrunner mit AndroidViewClient

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

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

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

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: Borho

Vorname: Simon

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

Qualitätssicherung in der Android App-Entwicklung durch Testautomatisierung

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

06.05.2021

Ort | Datum | Unterschrift im Original