Leave The House - Checklisten App

Studienarbeit

im Rahmen der Prüfung zum

Bachelor of Science (B.Sc.)

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Marius Huber

April 2021

Abgabedatum:	17. Mai 2021
Bearbeitungszeitraum:	01.10.2020 - 17.05.2021
Matrikelnummer, Kurs:	1286628, TINF18B2
Ausbildungsfirma:	SAP SE Dietmar-Hopp-Allee 16 69190 Walldorf, Deutschland
Gutachter der Dualen Hochschule:	Dr. Christian Bomhardt

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Studienarbeit mit dem Thema:

Leave The House - Checklisten App

gemäß § 5 der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. September 2017 selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Karlsruhe, den 10. Mai 2021

Huber, Marius

Abstract

- English -

This is the starting point of the Abstract. For the final bachelor thesis, there must be an abstract included in your document. So, start now writing it in German and English. The abstract is a short summary with around 200 to 250 words.

Try to include in this abstract the main question of your work, the methods you used or the main results of your work.

Abstract

- Deutsch -

Dies ist der Beginn des Abstracts. Für die finale Bachelorarbeit musst du ein Abstract in deinem Dokument mit einbauen. So, schreibe es am besten jetzt in Deutsch und Englisch. Das Abstract ist eine kurze Zusammenfassung mit ca. 200 bis 250 Wörtern.

Versuche in das Abstract folgende Punkte aufzunehmen: Fragestellung der Arbeit, methodische Vorgehensweise oder die Hauptergebnisse deiner Arbeit.

Inhaltsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
DHBW	Duale Hochschule Baden-Württemberg
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
UI	User Interface
UML	Unified Modelling Language
UX	User Experience

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einführung

Diese Arbeit beschreibt die Planung und Durchführung des Projekts „Leave The House - Checklisten App“im Rahmen der Studienarbeit der Duale Hochschule Baden-Württemberg (DHBW) Karlsruhe.
In diesem Kapitel werden die Motivation hinter der Anwendung so wie die vor Beginn der Arbeit festgelegte Aufgabenstellung beschrieben. Im weiteren Verlauf der Arbeit wird tiefer auf die Planungs- sowie die Durchführungsphase eingegangen und aufgetretene Probleme erläutert. Zum Schluss wird ein Fazit über die Gesamtheit des Projekts abgegeben und ein möglicher Ausblick zur Weiterführung beschrieben.

1.1 Motivation

Das altbekannte Problem: Man ist zu spät dran und muss dringend los, doch im Hinterkopf kommt immer wieder der Gedanke was vergessen zu haben... Sind die Fenster geschlossen, der Ofen abgestellt, ist alles eingepackt? Solche oder andere Aufgaben schwirren einem dann durch den Kopf. Hat man es endlich geschafft das Haus zu verlassen und steht vor dem Auto oder Fahrrad kommt wieder so ein Gedankenblitz, habe ich jetzt in all der Eile die Tür überhaupt abgeschlossen?
Solche Erfahrungen habe ich in der Zeit meines Studiums selbst häufiger erlebt und habe zu oft einen extra Weg zurück zur Tür meines Wohnheimzimmers gemacht nur um fest zu stellen, dass die Tür in den häufigsten fällen doch abgeschlossen ist. Die „Leave The House - Checklisten App“soll diesem Problem der Unsicherheit Abhilfe verschaffen. In dieser Mobile-App sollen Checklisten für jegliche Situationen in denen das Haus verlassen wird angelegt werden können. Die Checklisten enthalten jeweils alle Aufgaben die vor oder bei dem Verlassen des Hauses erledigt werden müssen. Sollte es wieder vorkommen, dass ein Gedanken des vergessen Habens vor dem Auto aufkommt, kann einfach in der App überprüft werden ob die Aufgabe erledigt wurde und sich so ein unnötiger Weg und das Gefühl etwas vergessen zu haben gespart werden.

1.2 Aufgabenstellung

Die Aufgabenstellung spiegelt den Kerngedanken hinter der App wieder.
Es soll eine iOS oder Android-App entwickelt werden. In dieser sollen anpassbare Checklisten angelegt werden können, die einen beim verlassen des Haus oder Arbeitsplatz unterstützen. Somit sollen Aufgaben die immer oder meistens beim verlassen eines Ortes auftreten in einer Checkliste erfasst und als erledigt markiert werden können.
Im folgenden Kapitel 2 Planung werden die in Abschnitt 1.1 Motivation genannten Anreize und die hier festgelegte Aufgabenstellung aufgegriffen und die daraus resultierte Projektplanung beschrieben.

2 Planung

Bevor mit der Durchführung eines Projekts angefangen werden kann, sollte mithilfe einer Projektdefinition das Projekt geplant werden. Dieses Kapitel beschreibt die Aspekte welche in der Projektdefinition für das Projekt festgelegt wurden. Unter anderem werden die Rahmenbedingungen, der Umfang so wie die Risikobehandlung erläutert.

2.1 Projektdefinition

In der Projektdefinition werden das Ziel, eine vereinfachte Strategie zur Erreichung des Ziels und der Bearbeitungszeitraum festgelegt.
Das Ziel ergibt sich in diesem Projekt aus der in Abschnitt 1.2 genannten Aufgabenstellung und lautet: „Entwicklung einer Android-App mit Checklisten Funktion“. Die Strategie wurde sehr simpel gehalten und ist: „Projektumfang planen und App entwickeln“. Diese vereinfachte Beschreibung der Strategie ist darauf zurückzuführen, dass es sich um ein Ein-Mann Projekt handelt und eine weitere Ausführung zur eventuellen Aufgabenteilung nicht als nötig erachtet wurde. Der Projektstart ist die offizielle Start der Studienarbeit der DHBW am 01.10.2020. Das Ende des Projektes spiegelt der Abgabetermin der Studienarbeit am 17.05.2021 wieder

2.2 Rahmenbedingungen

Mithilfe von Rahmenbedingungen werden die Grenzen des Projekts erstmals festgelegt. Gleichermaßen legen die Rahmenbedingungen grundlegende Entscheidungen fest, welche für die Durchführung des Projekts relevant sind. Die Rahmenbedingungen, welche in der Projektdefinition festgelegt wurden sind:

- Android Studio wird als Integrated Development Environment (IDE) verwendet
- Es wird ausschließlich eine Android und keine iOS-App entwickelt

- Das Projekt wird in der Studienarbeit beschreiben und von dem Betreuer Dr. Christian Bomhardt bewertet

Diese Rahmenbedingungen wurden aus bestimmten Gründen festgelegt, welche im folgenden erläutert werden. Der zweite Punkt grenzt die Entwicklung auf eines der zwei breit vertretenen Betriebssystemen für mobile Endgeräte ein. Hierbei wurde das Betriebssystem Android von Google festgelegt. Es wurde sich aus zwei Gründen bewusst gegen iOS entschieden. Der erste ist die Entwicklungsumgebung. Zur Entwicklung wird das von Apple entwickelte Programm Xcode benötigt. Um dieses zu herunterzuladen wird eine Apple-ID vorausgesetzt welche sich nun fließend mit dem zweiten Problempunkt in Verbindung bringen lässt. Die Apple-ID bezieht sich auf den eigene Apple-Account. Jeder der ein Apple Produkt besitzt hat in der Regel einen solchen Account. Da für dieses Projekt kein Budget zur Verfügung gestellt wird und weder Apple-Rechner noch mobile Endgeräte vorhanden sind kann weder eine App mit dem bereits genannten Programm Xcode entwickelt noch die App auf einen physischen Gerät getestet werden. Im Gegensatz zu iOS und Apple Produkten sind mehrere Android Geräte vorhanden welche zum testen unter realen Bedingungen genutzt werden können.[1][2]
Mit der Erläuterung des zweiten Punktes wird auch der erste deutlicher. Bei Android Studio handelt es sich um die von Android zur Verfügung Entwicklungsumgebung. Da Android als Grundlegendes Betriebssystem für die App festgelegt wurde ist Android Studio die logische Wahl. Zudem ist Android Studio frei verfügbar und lässt sich auch auf Windows-Rechnern installieren. Als Bonus wir es sogar mit der JetBrains Toolbox ausgeliefert, was die Installation und Aktualisierung des Programm weiter vereinfacht.[3]
Der dritte Punkt bezieht sich auf die Bewertung und den Abschluss des Projekts. Die Studienarbeit dient als Projektabschluss sowie als Grundlage zur Benotung des Projekts im Rahmen der Studienarbeit an der DHBW Karlsruhe. Bei der Studienarbeit handelt es sich um dieses Dokument. Die Benotung wird von dem Betreuer der Studienarbeit Dr. Christian Bomhardt vorgenommen.
Da die Rahmenbedingung jetzt ausführlich erläutert wurden wird im nächsten Abschnitt der Umfang der App ausgeführt.

[1] **chipIPhone.2019**.
[2] **Xcode.2021**.
[3] **AndroidStudio.2021**.

2.3 Umfang

Der Umfang legt die innerhalb des Rahmens zu erledigenden Aufgaben fest. Er ist sozusagen der Soll-Betrag, welcher zur Erreichung des Ziels benötigt wird. Im Zuge der App Entwicklung wurden hier die Anwendungsfälle festgelegt, welche in der App ausführbar sein müssen. Diese Anwendungsfälle sind:

- Erstellen einer Checkliste
- Bearbeiten einer Checkliste
- Löschen einer Checkliste
- Erstellen einer Aufgabe
- Bearbeiten einer Aufgabe
- Löschen einer Aufgabe
- Abhacken einer Aufgabe
- Hacken von einer Aufgabe entfernen
- Hacken von allen Aufgaben einer Checkliste entfernen

Die hier genannten Anwendungsfälle orientieren sich an der in Abschnitt 1.2 festgelegten Aufgabenstellung. Es soll dem Benutzer der App möglich sein in der Anwendung neue Checklisten anzulegen, um so für jede dem Nutzer nötige Situation eine Checkliste zur Verfügung zu haben. Da es sich um eine anpassbare Checkliste handeln soll, muss der Benutzer die Möglichkeit haben die Checkliste zu Bearbeiten. Im selben Zusammenhang sollte es dem Benutzer auch möglich sein nicht mehr benötigte Checklisten wieder zu löschen. Um die Checkliste nutzbar zu machen soll der Benutzer in der Lage sein Aufgaben in einer Checkliste zu erstellen. Im Sinne der Nutzererfahrung und Anpassbarkeit der Checkliste soll es ebenfalls möglich sein erstellte Aufgaben zu bearbeiten und auch wieder zu löschen. Die letzten drei Anwendungsfälle sind der Kern der Anwendung. Es muss dem Nutzer ermöglicht werden eine Aufgabe als erledigt zu markieren und diese Markierung auch wieder zu entfernen. In der Liste der Anwendungsfälle wird das Markieren beispielshaft als abhacken und entfernen des Hacken betitelt. Diese acht Anwendungsfälle stellen die Grundlage dar, welche die App erfüllen muss um als Nutzbar angesehen zu werden. Der neunte Punkt stellt eine zusätzliche Funktion dar, die dem Benutzer

die Handhabung erleichtern soll. Bei diesem Anwendungsfall sollen die „Hacken“aller Aufgaben innerhalb einer Checkliste entfernt werden. Dadurch muss der Benutzer nicht selbst alle Hacken entfernen wenn er die Checkliste wieder benutzen will und spart dadurch Zeit und Aufwand. Zusätzlich soll es bei dem Benutzer zu einer besseren Benutzererfahrung führen.
Nachdem die Anwendungsfälle geklärt sind wird nun weiter auf die Checkliste und Aufgabe an sich eingegangen. Diese stellen als Klasse die Modelle dieser Objekte dar. Abbildung 2.1 zeigt den Aufbau der Checkliste und der Aufgabe anhand eines Klassendiagramm. Sowohl Beide Klassen haben einen Titel und eine Beschreibung als String Attribut. Die Checkliste hält zusätzlich einen Array/Liste von Aufgaben. Die Aufgabe auf der anderen Seite hat als zusätzliches Attribut Abgehackt als Boolean. Die Checkliste hat neben den get- und set-Methoden jeweils eine Methode zum hinzufügen und entfernen von Aufgaben in das Aufgaben-Array Attribut. Die Aufgaben Klasse hat nur die get- und set-Methoden.

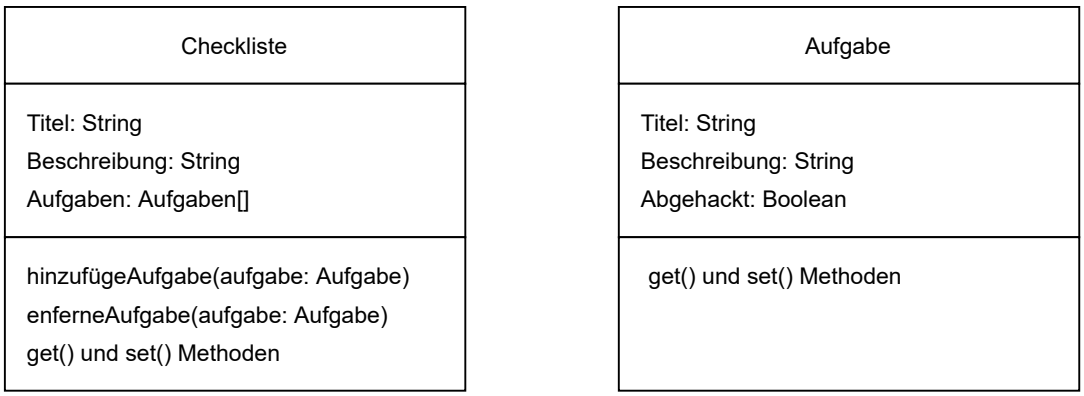

Abbildung 2.1: Klassen-Diagramm zu den Grundlegenden Klassen Checkliste und Aufgabe

Nach dem entwickeln und implementieren der Anwendungsfälle sollen diese getestet werden Hierfür sind im Umfang der Projektdefinition Espresso-UI Test vorgesehen. Mithilfe dieser Test können die Abläufe der Anwendungsfälle auf dem Emulator von Android Studio durchgeführt werden und so die Funktionalität gewährleistet werden. In der ursprünglichen Projektdefinition ist definiert, dass mindestens 80% der Anwendungsfälle mithilfe von Espresso Tests getestet werden sollen.
Als letztes legt die Definition des Umfang fest, dass eine Studienarbeit mit dem von der

DHBW Karlsruhe vorgegebenen Umfang von 40-60 Seiten geschrieben werden muss.

Nach Rücksprache mit dem Betreuer sind Ideen für eine Erweiterung des Umfang aufgetreten. Dabei handelt es sich um folgende Anwendungsfälle beziehungsweise Erweiterungen:

- Push-Benachrichtigungen bei dem verlassen einer für die Checkliste festgelegten Ort.
- Zeitstempel an den Aufgaben
- Historie

Der erste Punkt ist ein weiterer Anwendungsfall der verhindern soll, dass Nutzer vergessen die App und somit die Checkliste zu öffnen. Als Anhaltspunkt wurde die GeoFencing API genannt. Der zweite Punkt handelt von der Erweiterung der Aufgaben Klasse um ein Zeitstempel Attribut zu erweitern. Dieser den Fall vorbeugen, dass die Hacken vom vorherigen Tag noch vorhanden sind und der Nutzer so nicht weiß ob er diese Aufgaben am aktuellen Tag bereits erledigt hat. Der dritte Punkt spiegel eine Erweiterung des Zeitstempel als eigenen Anwendungsfall wieder. Hierbei soll die komplette Checkliste für einen festgelegten Zeitraum in einer Historie gespeichert werden. Dies soll dem Nutzer ermöglichen den Zustand der Aufgaben von vorherigen Tagen noch einmal einsehen zu können und sich so versichern zu können die Aufgabe erledigt zu haben.

2.4 Risikobehandlung

Zum Schluss des Planungskapitel und somit der ursprünglichen Projektdefinition wird das Thema Risikobehandlung behandelt. Die Risikobehandlung dient dem einschätzen und eingrenzen von Risiken. Dabei werden mögliche Risiken erfasst und beschrieben. Dazu zählen das Risiko, eine Beschreibung zu dem Risiko, eine geschätzte Wahrscheinlichkeit zu der das Risiko eintritt und eine Alternative mit welcher im Falle des Auftretens die Auswirkungen aufgefangen oder abgemildert werden soll. In Tabelle 2.1 wird die Risikobehandlung aus der Projektdefinition dargestellt.

Wie in Tabelle 2.1 zu sehen ist hält sich die Anzahl an Risiken in grenzen. Dies ist darauf zurückzuführen das nur eine Person und kein Team an dem Projekt arbeitet. Dadurch

Tabelle 2.1: Risikobehandlung

Risiko	Beschreibung	Wahrscheinlichkeit	Alternative
Verspäteter Projektstart	Die Bearbeitung des Studienarbeitsprojekt wird verspätet angefangen	80%	Verlorene Zeit wird zu einem späteren Zeitpunkt durch längere Arbeitstage und Wochenendschichten aufgeholt
Umfang nicht eingehalten	Aufgrund von Zeit oder Wissensmangel nicht alle Anwendungsfälle in der App implementiert	20%	Leistungsumfang im Bereich des möglichen verringern
Krankheit	Verminderter Fortschritt aufgrund von Krankheit	20%	Krankheit so gut wie möglich vermeiden, verlorene Zeit später aufholen
Vernachlässigte Dokumentation	Nicht alle Ideen und konkret geplante Umsetzungen vor der Implementierung dokumentiert	50%	Wenn möglich Dokumentation nachholen (Bsp. Diagramme)

können keine Kommunikationsprobleme auftreten. Ausfälle einzelner Personen und somit Nichterfüllung derer Aufgabenteile reflektieren hier das ganze Projekt und könnten somit alle mit den vier genannten Risiken abgedeckt werden.
Da es sich bei den in der Tabelle aufgeführten Risiken um die aus der Projektdefinition handelt werden eventuelle Eintritte und Erweiterungen im folgenden erläutert.
Das Risiko des verspäteten Projektstart ist wie anfangs korrekt eingeschätzt wurde eingetreten. Damit hat sich die Eintrittwahrscheinlichkeit von 80% bestätigt. Das Eintreten des Risiko ist auf eine schlechte Angewohnheit des Verantwortlichen zurückzuführen. Die Alternative zu diesem Risiko befindet sich in Ausübung und wird mit Ende der Ausarbeitung der Studienarbeit als erfolgreich angesehen.
Das zweit genannte Risiko konnte nach Projektdefinitionsstand erfolgreich verhindert werden. Das Risiko wird jedoch hiermit um „Erweiterungsumfang nicht eingehalten“erweitert,

da nach Rücksprache mit dem Betreuer Ideen für weitere Anwendungsfälle aufgetreten sind. Die Wahrscheinlichkeit für dieses angepasste Risiko wird somit im Nachhinein auf 60% angehoben. Die Alternative für Eintritt des Risiko wird beibehalten und vermutlich Anwendung für den Großteil der Ideen finden.
Die vernachlässigte Dokumentation ist ebenfalls in kleinerem Rahmen eingetreten. Die Studienarbeit selbst gilt als Hauptaspekt der Dokumentation und wird sicher fertiggestellt. Der eingetretene Bereich bezieht sich auf das erstellen von Unified Modelling Language (UML)-Diagrammen. Hier tritt ebenfalls die Alternative in Kraft und Diagramme werden nach Bedarf erstellt.
Das Risiko Krankheit konnte zu 100% vermieden werden.
In Folge des Kapitel 3 Durchführung werden unerwartete Probleme aufgeführt, welche Aufgrund des unerwarteten Eintritts keine Berücksichtigung in der Risikobehandlung gefunden haben. Diese werden im Abschnitt 3.4 Problembehandlung behandelt.

Mit Abschluss der Risikobehandlung gilt das Kapitel 2 Planung als abgeschlossen. In diesem Kapitel wurden die Projektdefinition welche zu Beginn des Projekts stattgefunden hat behandelt. Es wurde die konkrete Projektdefinition ausgeführt und mithilfe der Rahmenbedingungen und des Umfang eingegrenzt und konkretisiert. Als Abschluss wurde auf die Risikobehandlung eingegangen und eventuelle Eintritte und Änderungen erläutert.

3 Durchführung

Dieses Kapitel beschreibt die Durchführung des Projekts. Explizit wird hier der Beginn und Verlauf der Entwicklung der „Leave The House“-App behandelt. Zu Beginn wird der Start der Projektdurchführung, gefolgt von der eigentlichen Entwicklung beschrieben. Im Anschluss daran wird das Testen der Applikation behandelt. Zum Schluss dieses Kapitels werden in der Problembehandlung alle in den Anderen Abschnitten aufgetretenen Probleme ausführlich behandelt.

3.1 Projektdurchführung

Die Projektdurchführung behandelt den Start der eigentlichen Arbeit des Projekts. Wie bereits in Kapitel 2 beschrieben ist das Risiko des verspäteten Projektstarts eingetreten. Dies führte dazu, dass die Arbeit an dem Projekt nicht wie geplant am 01.01.2020 gestartet wurde. Der Projektstart begann stattdessen im März 2021. Durch die in der Risikobehandlung festgelegte Alternative hatte der verspätete Start keine, abgesehen der aus der Alternative resultierenden, Auswirkungen auf die Durchführung.
Die Bearbeitung des Projekts begann im März 2021 und endete mit Erreichung des geplanten Umfang Mitte April 2021. Durch die bereits genannte Erweiterung des Umfang nach Rücksprache mit dem Betreuer wird der endgültige Projektabschluss auf das Abgabedatum der Studienarbeit, den 17.05.2021, verlegt. Das Ende der Bearbeitung des zuvor geplanten Umfang wird weiterhin als Mitte April festgehalten. In der Verlängerten Bearbeitungszeit wird versucht die weiteren Ideen für den Umfang des Betreuer zu implementieren, da in dieser Zeit auch die Studienarbeit geschrieben werden muss kann eine vollständige Dokumentation dieser in der Studienarbeit nicht gewährleistet werden.

3.2 Entwicklung

In diesem Abschnitt wird der vollständige Verlauf der Entwicklung behandelt. Von der Erstellung des Projekts bis hin zum ersten fertigen Zustand der App. Die während der Entwicklung aufgetretenen Probleme werden hier genannt, jedoch erst im Abschnitt 3.4 Problembehandlung ausführlich behandelt. Das erstellen der im Umfang festgelegten Tests wird ebenfalls in einem anderen Abschnitt (3.3 Tests) behandelt.

3.2.1 Projekterstellung

Zu Beginn der Arbeit an einem Projekt muss zuerst das Projekt erstellt werden. Dies hängt damit zusammen, dass IDEs in der Regel Projekte als oberste Ordnerstruktur nutzen. Alle Dateien innerhalb des Projektordners können somit diesem Projekt zugeordnet werden. Die Entwicklung der App begann ebenfalls mit der Erstellung eines neuen Projekts in Android Studio. Dabei wird „New → New Project“ in der File Dropdown-Liste ausgewählt. Da Android nicht nur als Betriebssystem für Smartphones und Tablets sondern auch für Smartwatches oder in Autos und TVs verwendet wird muss als erster Schritt angegeben werden für welche Endplattform man eine Anwendung entwickeln möchte. In diesem Schritt kann auch direkt eine Projektvorlage ausgewählt werden. Aufgrund von geringer Erfahrung in der App-Entwicklung wurde für dieses Projekt die Vorlage „Basic Activity“(Basis Aktivität) anstatt einer „Empty Activity“(Leere Aktivität) als Grundlage für das Projekt gewählt. Im Gegensatz zu einer leeren Aktivität Vorlage beinhaltet die Basic Activity bereits ein Einstellungsmenü in der Werkzeugleiste am oberen Bildschirmrand, einem Knopf in der unteren Rechten Bildschirmecke und einen Knopf in der Mitte des Bildschirms, der einen Anzeigenwechsel bewirkt. Diese bereits vorhandenen Elemente und Funktionen erleichtern den Einstieg in die Entwicklung, da der Entwickler sich den Code dieser ansehen und somit leichter die Funktionsweise und Aufbau von Android-Apps verstehen kann. Hilfreich hierbei ist der in der Android Studio IDE eingebaute Emulator. Mit dem Emulator können virtuelle Android Geräte erstellt werden, um die App zu testen. Dabei kann die Android Version, so wie das Gerät ausgewählt werden. Der Emulator ist direkt zu Projektstart der Punkt an dem das erste Problem auftrat. Nach erstellen des Projekts kann man die gewählte Vorlage direkt über den Emulator testen. Dafür muss im AVD Manager ein neues Gerät erstellt werden, welches

zum testen genutzt werden soll. Bis hierhin lief alles reibungslos. Beim ausführen der App kam jedoch dann das Problem zum Vorschein, der Emulator konnte nicht gestartet werden. Es blieben nun nur zwei Möglichkeiten das Problem zu beheben, welche im Abschnitt 3.4 Problembehandlung behandelt werden. Nach wählen der Vorlage müssen noch weitere Informationen bei der Projekterstellung angegeben werden. Einerseits muss der Name des Projekts angegeben werden, hier wurde der Name der Endgültigen App „Leave The House“ eingegeben. Andererseits müssen grundlegende Entscheidung für die Entwicklung der App gefällt werden. Neben dem Namen muss noch die Programmiersprache und die minimal kompatible Anrdoid-Version gewählt werden. Android Apps können in zwei verschiedenen Sprachen geschrieben werden. Java und Kotlin. Für dieses Projekt wurde Kotlin als Programmiersprache für die App ausgewählt. Kotlin ist in der Android Entwicklung weit verbreitet und bietet eine Interoperabilität für Java. Somit kann während der App-Entwicklung auch auf Java zurückgegriffen werden, falls dies nötig sein sollte. Kotlin bietet im Gegensatz zu Java weitere Features wie null-Absicherung zurm Schutz vor NullPointer-Exceptions oder direkte View.Bindung.[1] Als minimale Android-Version wurde die Application Programming Interface (API) (Programmierschnittstelle) 23 festgelegt. Diese entspricht der Android Version 6.0 Marshmello und wird laut Android Studio zum aktuellen Zeitpunkt von 84.9% der Geräte unterstützt. Nach diesen Angaben wurde die Erstellung des Projekts in Android Studi erfolgreich abgeschlossen. An diesem Punkt (nach Behebung des Problems) könnte mit der Entwicklung begonnen werden, doch ein wichtiges weit verbreitetes und empfohlenes Mittel kann noch zum Projekt hinzugefügt werden. Die Versionsverwaltung.

Mithilfe von Versionsverwaltung können Änderungen an Dateien erfasst und verwaltet werden. Ein übliches Versionsverwaltungssystem ist der kostenlose Dienst GitHub. In GitHub können Nutzer für Projekte sogenannte „Repository“, zu Deutsch Verwaltungsorte, anlegen. Innerhalb eines Repository werden die Projektdateien verwaltet. GitHub erfasst jede Änderung an einer Datei und kann diese dem Nutzer anzeigen. Mithilfe eines commit können die Änderungen dann als neue Version im Repository abgelegt werden. Dies erlaubt es mehreren Nutzern an der gleichen Datei zu arbeiten ohne sich ständig in die Quere zu kommen. Außerdem erlaubt es dem Nutzer immer wieder auf stabile Versionen zurückzugreifen falls die aktuellen Änderungen nicht zum gewollten Ergebnis geführt haben. Im Zuge dieses Projektes wurde auf GitHub ein neues Repository angelegt, welches als Versionsverwaltung für die App genutzt wird. Damit können beispielsweise erfolgreiche

[1] **Kotlin.2020**.

Implementierungen eines Anwendungsfalls mithilfe eines commits in GitHub gesichert werden.
Mit der Erstellung des GitHub Repository und Verknüpfung des Android Studio Projekts damit ist die Projekterstellung abgeschlossen. Alle Vorbereitungen sind somit getroffen worden um eine Erfolgreiche Entwicklung zu gewährleisten.

3.2.2 Erstellen der Checkliste

Nach der erfolgreichen Projekterstellung begann die Arbeit an dem Projekt mit der Realisierung des ersten Anwendungsfall, dem erstellen einer Checkliste. Dafür wurde die Klasse Checkliste angelegt, diese wird in Listing 3.1 gezeigt. Die Klasse besteht aus einem initialen Konstruktor, welcher den Titel und die Beschreibung, sowie einem zweiten Konstruktor der neben Titel und Beschreibung noch eine Liste von Aufgaben entgegennimmt. Zudem enthält die Klasse, wie in Abschnitt 2.3 beschrieben, die Methoden um ein Element der Aufgaben Liste hinzuzufügen und um ein Element aus der Liste zu entfernen. Die ebenfalls beschrieben get() und set() Methoden sind in dieser Klasse nicht vorhanden, da Kotlin diese Methoden standardmäßig durch Zugriff auf die Variable zur Verfügung stellt. Um also auf den Titel einer Checkliste zuzugreifen genügt *checklist.title* anstelle von *checklist.getTitle*(). Nachdem das Modell zum halten der Checkliste, die Checkliste-Klasse erstellt wurde muss nun die Funktion implementiert werden ein Checklisten Objekt zu erstellen und auf dem Bildschirm anzuzeigen.

```
1  class Checklist(var title:String, var description: String?) {
2  var tasks = ArrayList<Task>()
3      constructor(title: String, description: String?, tasks: ↙
           ↳ ArrayList<Task>) : this(title, description){
4          this.tasks = tasks
5      }
6
7      fun addTask(task:Task){
8          tasks.add(task);
9      }
10
11     fun removeTask(task:Task){
```

```
12            tasks.remove(task);
13        }
14    }
```

Listing 3.1: Checkliste Klasse

Bevor diese Funktion jedoch implementiert werden kann muss der Ablauf dafür festgelegt werden. Der Nutzer soll auf den in der rechten unteren Bildschirmecke befindlichen Knopf drücken um eine neue Ansicht zu öffnen. Diese zeigt Eingabefelder für den Titel und die Beschreibung in denen der Benutzer diese Angaben tätigt, welche dann für den Konstruktor der Checklisten Klasse genutzt werden. Über einen Knopf an der gleichen Position wie der vorherige wird die Eingabe bestätigt und die Checkliste mit den Eingegeben Werten erstellt. Die erstellte Checkliste soll dann als Element einer Liste auf dem Bildschirm dargestellt werden. Anhand dieses Ablaufs wurde dann die Implementation dieser Funktion und Anwendungsfall begonnen. Abbildung 3.1 und Abbildung 3.2 zeigen die Layouts zu dem beschrieben Ablauf.
Android besteht Grundlegend aus Aktivitäten. Die Grundaktivität ist die sogenannte „Main Activity“. Diese bildet den Einstiegspunkt in die App und wird beim öffnen der App angezeigt. Eine Aktivität besteht meistens aus einer “Controller-“ und einer Layout Datei. In der Layout-Datei wird definiert was auf dem Bildschirm angezeigt wird wenn die Aktivität ausgeführt wird. Dabei handelt es sich um eine XML-Datei in der Elemente wie Knöpfe, Text oder weitere mithilfe eines Layouts angeordnet werden können. Die Controller-Datei spiegelt dagegen die funktionale Ebene der Aktivität wieder. In ihr werden Funktionen ausgeführt und der Aktivitäts-Lebenszyklus behandelt. Eine Aktivität hat einen Lebenszyklus der aus verschiedenen Zuständen besteht. Der erste Zustand der ausgeführt wird ist „onCreate()“. In dieser Methode wird angegeben welches Layout dem Nutzer angezeigt werden soll. Diese Methode wird in der Regel immer überschrieben, um mittels setContentView() das Layout anzugeben. Listing 3.3 zeigt die onCreate-Methode der Aktivität CreateChecklist. Hier wird wie angegeben die onCreate-Methode überschrieben und über setContentView das zugehörige Layout zur Darstellung auf dem Bildschirm angegeben. Zudem wird über die seSupportActionBar-Methode die im Layout definierte Werkzeugleiste als SupportActionBar festgelegt. Damit wird der Aktivität ermöglicht die Interaktion mit eventuell in der Werkzeugleiste vorhanden Knöpfen zu erfassen und entsprechende Funktionen auszuführen. In der weiteren Beschreibung zur Implementierung des Anwendungsfall zum erstellen einer Aktivität wird weiter auf das

Codebeispiel eingegangen. Die weiteren Zustände des Lebenszyklus sind onStart() (Startet die Aktivität und macht sie sichtbar und ermöglicht Interaktivität), onResume() (Wird ausgeführt wenn die Aktivität wieder Interagierbar wird), onPause() (Dieser Zustand tritt auf wenn die Aktiviät den Fokus verliert und ist oft ein Indikator dass die Aktivität verlassen wird), onStop() (Die Aktivität ist nicht länger sichtbar für den Nutzer), onRestart(Wechsel von onStop() in onStart()) und onDestroy() (Zerstört die Aktivität und endet den Lebenszyklus).[2] Die Grundlegende Funktionsweise von Aktivitäten sollte nun verstanden worden sein.
Um den Anwendungsfall eine Checkliste erstellen zu realisieren wird zunächst eine neue Aktivität erstellt. Dazu wird über File → New → Activity → Empty Activity eine neue Aktivität angelegt. Hierbei muss der Name der Aktivität und des Layout angegeben werden. Der Name des Layout wird von Android Studio passend zu der Eingabe im Aktivitätsnamen-Feld angepasst. Auch hier kann die Programmiersprache gewählt werden. Das lässt sich auf die Interoperabilität zurückführen, da damit auch eine Java-Aktivität in einer Kotlin Anwendung ausgeführt werden kann.
In der onCreate-Methode wird wie bereits verdeutlicht das Layout für die Aktivität festgelegt. Das für die CreateActivity erstellte Layout kann in Abbildung 3.2 eingesehen werden. Es stellt jeweils ein Label sowie ein Eingabefeld für den Titel und die Beschreibung dar. Zusätzlich befindet sich in der unteren rechten Ecke ein Knopf über den die Checkliste mit den Eingaben aus den Eingabefeldern hinzugefügt werden soll.
Um dieses Layout auf dem Bildschirm zu sehen muss zunächst die dazugehörige Aktivität gestartet werden. Wie bereits erläutert wird die App mit der MainActivity gestartet, welche als Einstiegspunkt für die Anwendung dient. Das Layout der MainActivity kann in Abbildung 3.1 angesehen werden. Dieses Layout besteht aus einer Werkzeugleiste, der bereits im Ablauf für den Anwendungsfall beschriebene Liste in Form einer RecyclerView zum Anzeigen der Checklisten und dem aus der Vorlage übernommenen und angepassten Knopf. Mit diesem Knopf soll der Anwendungsfall und die createChecklist Aktivität gestartet werden. Um dem Knopf die Funktionalität dafür zu geben wird er zunächst über die findViewById-Methode der Knopf mithilfe der ID gefunden, um so mit ihm innerhalb dieser Klasse interagieren zu können. Für die Interaktion wird der Knopf mit einem onClickListener versehen. Dieser führt die darin geschriebene Funktion aus sobald der Nutzer den Knopf betätigt. In diesem Fall soll die CreateChecklist Aktivität gestartet werden. Dazu wird ein Intent deklariert. Ein Intent ist ein Nachrichten-Objekt das genutzt wird

[2] **AktivitÃd̈ten.2021**.

um Aktionen von einer anderen Anwendungskomponente anzufragen. Das starten einer Aktivität stellt einen der drei fundamentalen Anwendungsfälle des Intent Objekts dar.[3] Der Codeausschnitt Listing 3.2 zeigt die Deklaration des Intent un der darauffolgende Aufruf zum starten der CreateChecklist Aktivität im onClickListener des Knopf. Bei der Initialisierung des Intent muss die jeweilig zu startende Aktivität als Parameter übergeben werden. Diese wird im Anschluss mit dem Befehl startActivity() oder in diesem Fall startActivityForResult() durch Übergabe des Intent in den Befehl gestartet. Zusätzlich wird beim Starten der Aktivität ein weiterer Parameter übergeben. Dieser stellt einen requestCode dar mit dessen Hilfe Ergebnisse von Aktivitäten unterschieden werden können. Das ermöglicht den korrekten Umgang mit den im Ergebnis potentiell übergebenen Daten.

```
1 findViewById<FloatingActionButton>(R.id.add_checklist).setOnClickListener ↙
    ↳ { view ->
2             val intent = Intent(this, ↙
                ↳ CreateChecklist::class.java)
3             startActivityForResult(intent, ↙
                ↳ Finals.CREATE_CHECKLIST)
4         }
```

Listing 3.2: Start der CreateChecklist Aktivität

Nachdem die Aktivität gestartet wurde wird, wie mit dem Aktivitäts-Lebenszyklus beschreiben die onCreate-Methode der CreateChecklist Aktivität ausgeführt. Wie in Listing 3.3 zu sehen wird dort ebenfalls der Knopf mit einem onClickListener versehen. Dieser beendet im Gegensatz zum anderen die Aktivität anstatt eine neue MainActivity zu starten. Allerdings wird ebenfalls ein Intent erstellt. Mithilfe des Intent werden die Nutzereingaben der MainActivity als Ergebnis der createActivity Aktivität übergeben. Um die Nutzereingaben zu lesen werden die Textfelder, wie die Knöpfe zuvor, mithilfe der ID gefunden un der deren Text-Attribut ausgelesen. Die putExtra-Methode des Intent erlaubt es über einen Intent zusätzliche Daten zu übergeben. Dazu muss in dieser Methode ein Schlüssel-Wert Paar erstellt werden. Nachdem die Nutzereingaben dem Intent beigefügt wurden wird dieser in der setResult-Methode zusammen mit einem RequestCode übergeben. Der RequestCode spiegelt hier den gleichen wieder der auch zum starten der Aktivität übergeben wurde. Die zwei darauffolgenden Methodenaufrufe dienen dem

[3] **Intents.2021**.

Beenden der Aktivität. Das Ergebnis wird durch die onActivityResult-Methode weiter verarbeitet, welche in Listing 3.4 teilweise einsehbar ist. Diese wird, wie die onCreate-Methode, überschrieben um eigenen Code ausführen zu können. Die Methode bekommt als Parameter einen RequestCode, einen resultCode und einen Intent als Datenhalter übergeben. Hier wird der Nutzen des requestCode deutlich. Er wird als Schlüsselvariable für den Switch genutzt, um Ergebnisse aus unterschiedlichen Aktivitäten unterscheiden zu können. Für den Aktuellen Stand von nur einer Aktivität wäre der Switch nicht notwendig, da im weiteren verlauf der Entwicklung weitere hinzukommen wurde er bereits von Anfang an implementiert. Der resultCode stellt den Status des Ergebnis der Aktivität dar und könnte beispielsweise Result.OK lauten. Dieser findet hier allerdings keine Verwendung da die Ergebnisüberprüfung über ein Extra des Intent behandelt wird. Bei dem Intent handelt es sich um den in der setResult-Methode übergebenen Intent. Zu Beginn der Bearbeitung der vom Intent übergebenen Daten wird über das „successful“-Extra geprüft ob das übergeben der Daten erfolgreich war. Im Anschluss wird ein neues Checklisten-Objekt erstellt. Dieses erhält als Parameter die vom Nutzer in der CreateChecklist Aktivität eingegebenen Titel und Beschreibung. Im Anschluss daran wird überprüft ob der Eingegebene Titel ein Duplikat ist. Falls dem so ist wird die neue Checkliste nicht der Liste von Checklisten zugefügt, welche als Datenhalter in der Main-Aktivität dient, und eine Fehlermeldung auf dem Bildschirm angezeigt. Falls nicht wird die neue Checkliste der Liste hinzugefügt und infolgedessen auf dem Bildschirm angezeigt.

```
1  override fun onCreate(savedInstanceState: Bundle?) {
2          super.onCreate(savedInstanceState)
3          setContentView(R.layout.activity_create_checklist)
4          setSupportActionBar(findViewById(R.id.toolbar))
5
6          val createChecklistFab = ↙
               ↳ findViewById<FloatingActionButton>(R.id.createChecklistFab)
7
8          createChecklistFab.setOnClickListener{
9              val title = ↙
                   ↳ findViewById<EditText>(R.id.titleInput).text.toString()
```

```
10              val description = ↙
                    ↳ findViewById<EditText>(R.id.descriptionInput).text.toStrin
11
12              intent = Intent()
13              intent.putExtra("successful", true)
14              intent.putExtra("title", title)
15              intent.putExtra("description", description)
16
17              setResult(Finals.CREATE_CHECKLIST, intent)
18              finishActivity(Finals.CREATE_CHECKLIST)
19              finish()
20          }
21      }
```

Listing 3.3: onCreate Methode der CreateChecklist Aktivität

```
1  override fun onActivityResult(requestCode: Int, resultCode: ↙
       ↳ Int, data: Intent?) {
2          super.onActivityResult(requestCode, resultCode, data)
3          when(requestCode){
4              Finals.CREATE_CHECKLIST -> {
5                  val bundle = data!!.extras
6                  if (bundle!!.get("successful") as Boolean) {
7                      val newChecklist = Checklist(
8                          bundle.get("title") as String,
9                          bundle.get("description") as String
10                     )
11                     var uniqueTitle: Boolean = true
12                     for (i in this.data) {
13                         if (i.title == newChecklist.title) {
14                             uniqueTitle = false
15                         }
16                     }
17                     if (uniqueTitle) {
18                         this.data.add(newChecklist)
19                     } else {
20                         Snackbar.make(
```

```
21                                  findViewById(R.id.coordinaterLayoutMain),
22                                  R.string.duplicateTitle,
23                                  Snackbar.LENGTH_SHORT
24                              ).show()
25                          }
```

Listing 3.4: onActivityResult Methode der Main Aktivität

Die Checklisten sollen in Form einer Liste auf dem Bildschirm dargestellt werden. Dazu wird wie im Layout der MainActivity beschrieben eine RecyclerView genutzt. Im Vergleich zu einer herkömmlichen ListView ist die RecyclerView eine modernere, flexiblere und performantere Alternative. Im Gegensatz zu Knöpfen und Textfeldern kann die RecyclerView nicht einfach mithilfe der ID gefunden und beispielsweise der Text gesetzt werden. Um Einträge in der RecyclerView anzuzeigen wird ein Adapter benötigt. Das ist Notwendig da jedes Item in der Liste als separate View behandelt wird und gesondert dargestellt werden muss. Um die Checklisten in der RecyclerView darstellen zu können wurde der ChecklistRecyclerViewAdapter als Klasse angelegt. Dieser erhält als Parameter die Liste von Checklisten aus der MainActivity, da diese die Informationen zu den Items hält die Dargestellt werden sollen. Zusätzlich wird innerhalb dieser Klasse eine weitere Klasse erstellt. Diese Klasse ist ein ViewHolder und heißt ChecklistViewHolder. Jedes Item in der RecyclerView wird durch eine Instanz dieses ViewHolder definiert. Für jeden Eintrag in der Liste wird vom Adapter ein ViewHolder erstellt und im Anschluss daran die Daten daran gebunden. Listing 3.5 zeigt die Methoden onCreateViewHolder und onBindViewHolder des ViewHolder. Ähnlich wie bei den Aktivitäten wird in der onCreate-Methode das Layout festgelegt. Bei dem Layout handelt es sich um ein speziell für die Liste erstelltes Layout, welches einen Eintrag der Liste repräsentiert. In Abbildung 3.1 können drei dieser Listenitems gesehen werden. Diese bestehen aus einem Textfeld für den Titel der Checkliste und einem weiteren, etwas nach rechts gerücktem, mit kleinerer und grauer Textfont versehenen Textfeld für die Beschreibung. Die onBindViewHolder-Methode setzt den Text dieser Textfelder, sodass die richtigen Daten angezeigt werden. In der MainActivity wird um die Daten darzustellen die RecyclerView anhand der ID gefunden und eine Instanz des Adapter erstellt. Dabei wird die Liste der Checkliste als darzustellende Items übergeben. Damit frisch erstellte Checklisten in der Liste angezeigt werden wurde die onActivityResult-Methode um dem Funktionsaufruf $checklistRecyclerView.adapter?.notifyDataSetChanged()$ erweitert. Dieser teilt dem

Adapter mit, dass sich die Daten verändert haben und veranlasst so ein neu laden der Liste.

```
1  override fun onCreateViewHolder(parent: ViewGroup, viewType: ↙
       ↳ Int): ChecklistViewHolder {
2      val layoutInflater =  LayoutInflater.from(context)
3      var view: View = ↙
           ↳ layoutInflater.inflate(R.layout.checklist_row, ↙
           ↳ parent, false)
4
5      return ChecklistViewHolder(view, onChecklistListener)
6  }
7
8  override fun onBindViewHolder(holder: ChecklistViewHolder, ↙
       ↳ position: Int) {
9      holder.titleTextView.text = checklists[position].title
10     holder.descriptionTextView.text = ↙
           ↳ checklists[position].description
11     holder.editMode = this.editMode
12 }
```

Listing 3.5: ViewHolder Methoden des ChecklistRecyclerViewAdapter

Mit dem darstellen der erstellten Checkliste kann der erste Anwendungsfall, das erstellen einer Checkliste, nahezu als erfolgreich implementiert angesehen werden. Es ist dem Nutzer möglich über den Knopf die CreateChecklist Aktivität zu starten und seine Eingaben für Titel und Beschreibung zu tätigen. Durch bestätigen der Eingaben wird das neue Checklisten Objekt erstellt und dynamisch in der RecyclerView-Liste eingefügt und dargestellt. Der letzte fehlende Schritt ist das speichern und laden von erstellten Checklisten. Dieser Schritt wird in Unterabschnitt 3.2.3 erläutert.

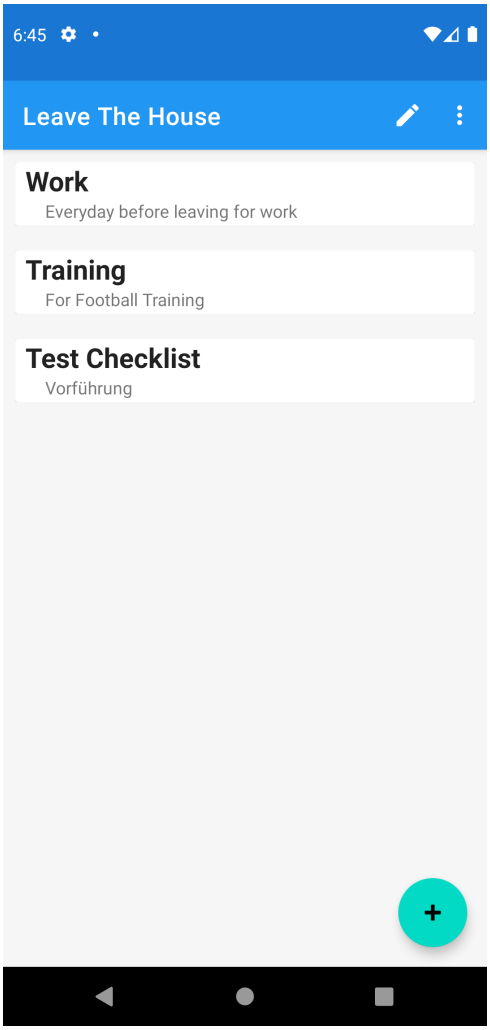

Abbildung 3.1: MainActivity Ansicht

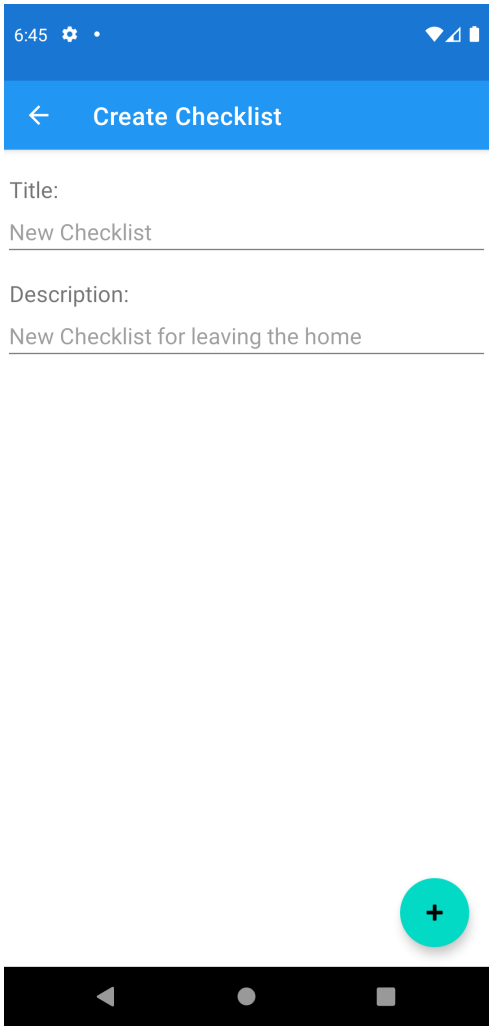

Abbildung 3.2: CreateChecklist Ansicht

3.2.3 Speichern von Checklisten

Nachdem es dem Nutzer möglich ist Checklisten zu erstellen ist der nächste Schritt die erstellte Checkliste zu speichern und beim öffnen der App die gespeicherten Checklisten wieder zu laden und in der RecyclerView anzuzeigen.
Die erste Überlegung war hierfür ein Backend zu entwickeln, welches die Daten in einer Datenbank speichert und über eine API ansprechbar ist. Voraussetzung für diese Herangehensweise wäre ein Nutzer-System. Jeder Nutzer müsste einen Account erstellen und diesen mit einem Passwort absichern. Ohne das könnten die gespeicherten Checklisten nicht dem richtigen Benutzer zugeordnet werden. Im verlauf der Entwicklung wurde sich

gegen diese Herangehensweise entschieden und stattdessen das Speichern der Daten in einer Datei auf dem gerät gewählt. Grund dafür sind teilweise persönliche Präferenzen und Abwägung an den Vorteilen, die eine solche Lösung mit sich bringen würde. Auf der einen Seite würde es dem Nutzer ermöglichen Speicher auf dem Endgerät zu sparen. Auf der anderen Seite jedoch müsste ein Nutzer sich einen Account erstellen, bei dem er eine valide E-Mail Adresse, eventuell einen Nutzernamen und ein Passwort festlegen nur um Checklisten erstellen zu können. Zusätzlich müsste die Anwendung eine Internetverbindung haben und der Nutzer somit der App diesen Zugriff auf seinem gerät erlauben. Diese Punkte spiegeln auch die persönlichen Ablehnungen für diese Herangehensweise wieder, da ich dem erstellen von Accounts für skeptisch gegenüberstehe und dabei auch schon schlechte Erfahrungen mit Veröffentlichung von E-Mail Adressen nach Angriffen auf die Server gesammelt habe. Zudem sollte man für das erstellen und speichern einer Checkliste keine Internetverbindung brauchen, wobei dieser Punkt in der heutigen zeit zu vernachlässigen ist da jedes Endgerät, zumindest Zuhause per WLAN, eine Internetverbindung hat. Für diese App wurde sich jedoch für das lokale Speichern der Daten in einer Datei entschieden. Android unterstützt diese Herangehensweise durch das interne Datei System. Dies ermöglicht es Dateien Lokal auf dem Gerät persistent zu speichern. Als Format wurde hierfür die JSON gewählt. Bei JSON handelt es sich um ein Dateiformat, welches Programmiersprachen unabhängig und weit verbreitet ist. Die JSON-Codierung ist zudem eine einfache Textform und lässt sich dadurch einfach zusammensetzten.
Das Speichern der Daten in der App erfolgt in einer dafür selbst geschriebenen Methode, um das Speichern unabhängig von einer spezifischen Standard Android Methode ausführen zu können. Diese Ansatz ermöglicht es die Speicher-Methode an den Stellen aufzurufen an denen es sinnvoll und notwendig ist. Listing 3.6 zeigt die geschriebene Methode und Listing 3.7 zeigt den Aufbau des JSON-Format in dem die Checklisten gespeichert werden.

```
1 private fun saveChecklistsToFile(context: Context){
2
3     var checklistsString: String = "{\n"
4
5     for (i in this.data.indices){
6         val checklistJson = JSONObject()
7         checklistJson.put("title", this.data[i].title)
```

```
8           checklistJson.put("description", ↙
                ↳ this.data[i].description)
9           var checklistString = checklistJson.toString()
10
11          checklistsString += "\"$i\": $checklistString"
12          if(i + 1 != this.data.size){
13              checklistsString += ","
14          }
15      }
16      checklistsString += "\n}"
17
18      var file = File(context.filesDir, FILENAME)
19      var fileWriter = FileWriter(file)
20      var bufferedWriter = BufferedWriter(fileWriter)
21      bufferedWriter.write(checklistsString)
22      bufferedWriter.close()
23  }
```

Listing 3.6: Methode zum Speichern von Checklisten

```
1   {
2       "1": {
3           "title": "Checklist Title",
4           "description": "Checklist Description"
5       },
6       "2": {
7           "title": "Test Title",
8           "description": "Test Description"
9       }
10  }
```

Listing 3.7: JSON Format zum Speichern der Checklisten

Die Methode baut einen String im JSON-Format. Dazu wird über die Liste der Checklisten iteriert und das Checklisten-Objekt in ein JSON-Objekt konvertiert. Nach dieser Konvertierung wird das JSON-Objekt in einen String umgewandelt und nach weiterer Formatierung mit dem gesamt String über String-Konkatenation zusammengeführt. Nachdem alle Checklisten formatiert und zu einem String geformt wurden muss dieser

„JSON-String“ noch in einer Datei auf dem Gerät gespeichert werden. Dazu wird ein Datei-Objekt erstellt welches als Parameter das Lokale Dateiverzeichnis und den Dateinamen übergeben bekommt. Mithilfe eines Buffered- und eines FileWriters wird nun der String in die angegebene Datei in dem angegebene Verzeichnis geschrieben. Falls die Datei noch nicht existiert wird Sie erstellt und falls sie existiert wird der Inhalt überschrieben. Das überschreiben der vorhandene Datei ist vor allem für das Bearbeitern und Löschen für Checklisten relevant. Die Speichern-Methode wird an zwei Stellen in der MainActivity aufgerufen um zu gewährleisten, dass die Checklisten gespeichert werden. Diese zwei Stellen sind die Android-Methoden onStop() und onBackPressed(). Die onStop-Methode stammt vom Aktivitäten-Lebenszyklus und wird ausgeführt sobald die Aktivität in den Stopp-Status wechselt. Die onBackPressed-Methode erkennt wenn der Nutzer die Zurück-Taste auf dem Gerät betätigt um die Anwendung zu verlassen. Mithilfe dieser beiden Methoden sollte somit das Speichern der Checklisten gewährleistet sein.
Nachdem die Checklisten nun in einer lokalen Datei auf dem gerät gespeichert sind müssen diese Daten beim starten der App wieder eingelesen und dargestellt werden. Das Laden wird analog zu der Speichern-Methode in einer separaten Methode ausgeführt. Im Gegensatz zum Speichern wird hier zu Beginn der Methode mithilfe eines FileReaders und StringBuilders aus der JSON-Datei wieder ein String im JSON-Format erzeugt. Die einzelnen JSON-Objekte werden dann aus dem String ausgelesen und aus den Daten ein Checklisten-Objekt erzeugt, welches der Checklisten-Liste hinzugefügt wird. Die Laden-Methode wird einmal in der onCreate-Methode der Main-Aktivität aufgerufen. Der Aufruf erfolgt vor dem erstellen des Adapter wodurch dieser zum Start der App die geladenen Daten darstellen kann ohne mit notifyDataSetChanged() ein neu generieren der RecyclerView Items anfragen zu müssen.
Mit diesen zwei Methoden kann der Anwendungsfall „Checkliste erstellen“ als vollständig abgeschlossen angesehen werden. Der Nutzer kann nun Checklisten erstellen, welche sowohl direkt nach dem Erstellen als auch nach starten der App angezeigt werden.

3.2.4 Erstellen von Aufgaben

Der zweite Anwendungsfall der im Verlauf der Entwicklung implementiert wurde ist das Erstellen von Aufgaben zu einer Checkliste. Hierzu soll der Nutzer auf eine Checkliste in der RecyclerView tippen können, um so diese Checkliste zu öffnen. Die Ansicht der

geöffneten Checkliste soll dann die Möglichkeit bieten eine Aufgabe zu erstellen und diese ebenfalls in einer RecyclerView-Liste anzuzeigen.
Der erste Schritt ist das dynamische öffnen des an getippten Checklisten-Listenelement. Dazu wird der ChecklistViewHolder um eine Vererbung von der OnClickListener-Klasse erweitert. Dadurch kann die onClick-Methode innerhalb des ViewHolder überschreiben und mit eigenen Methodenaufrufen ausgestattet werden. Durch die generelle Funktionsweise des Adapter und ViewHolder ist somit jedes Listenelement mit dieser Funktion ausgestattet und kann auf die Interaktion des Nutzer reagieren. Für das öffnen der Checkliste wurde die neue Aktivität OpenChecklistActivity erstellt. Der Layout-Aufbau gleicht dem der MainActivity. Es besteht ebenfalls aus einer Werkzeugleiste, einer RecyclerView und einem Knopf in der unteren rechten Ecke. Im Gegensatz zur MainActivity steht jedoch nicht der Name der App sondern der Titel der geöffneten Checkliste als Titel in der Werkzeugleiste. Dies ermöglicht es dem Nutzer zu erkennen welche Checkliste er aktuell geöffnet hat. Innerhalb der onClick-Methode wird die OpenChecklistActivity durch einen Aufruf von startActivity gestartet. Das Layout kann in Abbildung 3.3 eingesehen werden. Der Knopf unten rechts erfüllt die gleiche Aufgabe wie sein äquivalent in der MainActivity. Mit ihm wird die CreateTask Aktivität gestartet in der der Nutzer der Aufgabe einen Titel und eine Beschreibung geben kann (vgl. Abbildung 3.4). Diese werden wieder in der onActivityResult-Methode ausgewertet und ein neues Aufgaben Objekt damit erstellt. Die Aufgaben werden dann mithilfe eines TaskRecyclerViewAdapter und der darin erstellten Klasse TaskViewHolder in der RecyclerView dargestellt. Der letzte fehlende Schritt ist das Speichern und Laden der Aufgaben zu der jeweiligen Checkliste. An dieser Stelle ist ein Problem mit dieser Umsetzung aufgetreten. Die Aufgaben wurden in der openChecklistAktivität erstellt und wie die Checklisten in der MainActivity in einer ArrayList gehalten. Die Speichermethode befindet sich jedoch in der MainActivity da nur dort eine Liste mit allen Checklisten vorhanden ist. Es gibt jedoch keine Möglichkeit die Aufgabenliste über den Adapter an die MainActivity zu geben. Wie dieses Problem gelöst werden konnte wird im Abschnitt 3.4 erläutert.

Nach der Lösung des Problems fehlt noch das Speichern und Laden der Aufgaben. Dazu wurden die dafür geschriebenen Methoden angepasst. Der angepasste Code kann in Listing 3.8 eingesehen werden. Um die Aufgaben zu Speichern wird ein JSONArray-Objekt erstellt. Jede Aufgabe wird dann, wie die Checklisten auch, in ein JSON-Objekt umge-

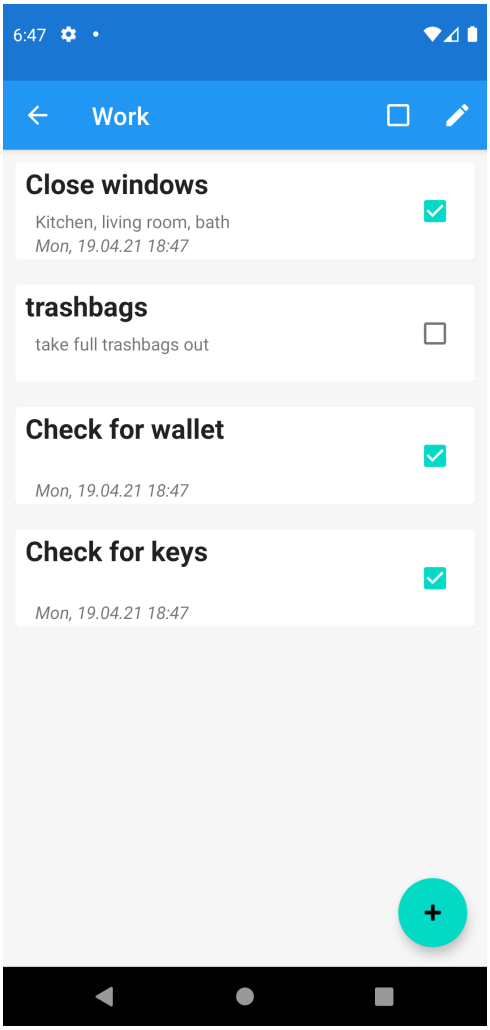

Abbildung 3.3: Ansicht einer geöffneten Checkliste

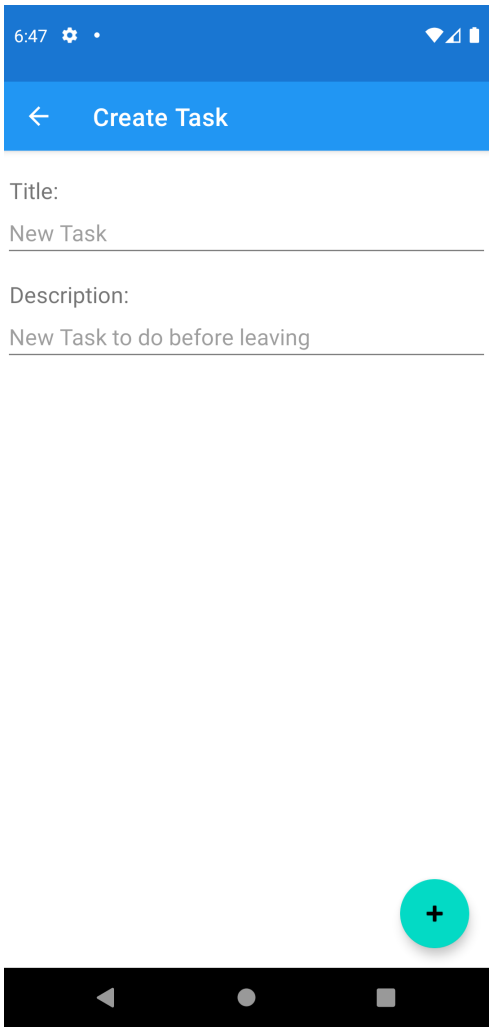

Abbildung 3.4: CreateTask Ansicht

wandelt und an den JSONArray angehängt. Nachdem alle Aufgaben als JSON-Objekte an den JSONArray angehängt wurde wird dieser dem JSON-Objekt der Checkliste beigefügt. Der Rest der Methode bleibt unverändert. Im Anschluss an die Erstellung des JSON-String wird über den FileWriter die JSON-Datei geschrieben und auf dem Gerät im lokalen Speicher gesichert. Das neue resultierende JSON-Format wird in Listing 3.8 beispielhaft gezeigt.

```
1       var checklistsString: String = "{\n"
2
```

```
3       for (i in this.data.indices){
4           val checklistJson = JSONObject()
5           checklistJson.put("title", this.data[i].title)
6           checklistJson.put("description", ↙
                ↳ this.data[i].description)
7           var tasksJson = JSONArray()
8           for (j in this.data[i].tasks.indices){
9               var taskJson = JSONObject()
10              taskJson.put("title", this.data[i].tasks[j].title)
11              taskJson.put("description", ↙
                    ↳ this.data[i].tasks[j].description)
12              taskJson.put("checked", ↙
                    ↳ this.data[i].tasks[j].checked)
13              taskJson.put("lastChecked", ↙
                    ↳ this.data[i].tasks[j].lastChecked)
14              tasksJson.put(taskJson)
15          }
16          checklistJson.put("tasks", tasksJson)
17          var checklistString = checklistJson.toString()
18          checklistsString += "\"$i\": $checklistString"
19          if(i + 1 != this.data.size){
20              checklistsString += ","
21          }
22      }
23      checklistsString += "\n}"
```

Listing 3.8: Methode zum Speichern von Checklisten und Aufgaben

```
1  {
2      "1": {
3          "title": "Checklist Title",
4          "description": "Checklist Description",
5          "tasks": [
6              {
7                  "title": "Task Title",
8                  "description": "Task Description",
9                  "checked": false
```

```
10              },
11              {
12                  "title": "Test Task Title",
13                  "description": "Test Task Description",
14                  "checked": false
15              }
16          ]
17      },
18      "2": {
19          "title": "Test Title",
20          "description": "Test Description",
21          "tasks": []
22      }
23  }
```

Listing 3.9: JSON Format zum Speichern der Checklisten und Aufgaben

Mit diesen Änderungen ist der Anwendungsfall Aufgabe erstellen abgeschlossen. Während der Entwicklung des Anwendungsfall wurde zugleich der Anwendungsfall des abhacken von Aufgaben implementiert. Hierzu wurde der TaskViewHolder, gleich wie der Checklist-ViewHolder, um einen onClickListener erweitert. Darin wird eine Methode aufgerufen die Abhängig vom aktuellen Status in den jeweils anderen gewechselt. Hier wird jetzt ein Unterschied des Layout eines Aufgaben-Listenelement zum bereits erläuterten Checklisten-Listenelement deutlich. Dieses hat neben dem Titel- und Beschreibungstextfeld zusätzlich eine Checkbox auf der rechten Seite. Über diese Checkbox wird der aktuelle Staus der Aufgabe repräsentiert. Eine abgehackte Checkbox gilt als erledigt, eine leere als offen. In Abbildung 3.3 sind die Checkbox so wie die beiden möglichen Zustände deutlich zu erkennen. In der onClick-Methode wird neben dem Wert des Aufgaben-Objekt auch der Wert der Checkbox im jeweiligen Layout des an getippten ViewHolder-Elements gewechselt.
Durch diese kleine Ergänzung am Layout der Aufgaben und dem ViewHolder kann auch dieser Anwendungsfall als erfolgreich implementiert angesehen werden.

3.2.5 Bearbeiten von Checklisten und Aufgaben

Als nächstes wurden die Anwendungsfälle des Bearbeiten einer Checkliste und einer Aufgabe implementiert. Die Idee hierbei ist den gleichen Ablauf für Checklisten und Aufgaben zu verwenden, wie es auch beim Erstellen der Fall ist. Das Erstellen läuft in beiden Fällen über den Knopf in der rechten unteren Ecke der nahezu identische Aktivitäten für die Benutzereingaben öffnet. Die Wahl die Abläufe gleich zu implementieren lässt sich auf User Experience (UX), zu deutsch Nutzererfahrung zurückführen. Wenn gleiche Funktionen innerhalb einer Anwendung an verschiedenen Stellen zu finden sind löst dies bei einem Benutzer Unbehagen aus. Durch verwenden einer Funktion versteht der Nutzer den Ablauf. Versucht er nun eine gleiche Funktion an einer anderen Stelle der gleichen Anwendung durchzuführen wird er Versuchen auf seine Erfahrung des vorherigen Ablauf zurückzugreifen. Das beruht auf dem grundlegenden Verständnis dass gleiche Funktionen den gleichen Ablauf haben sollte. Ein gutes Beispiel hierfür ist das sogenannte Norman-Tür Prinzip. Bei einer Norman-Türe handelt es sich um jede Türe deren Benutzung verwirrend oder schwierig ist. Türen haben an sich einen einfach Anwendungsfall der sich in der Durchführung lediglich durch den Türgriff unterscheidet. Im Lauf seines Lebens lernt man das eine Tür mit Griff aufgezogen wird, während man eine Tür die nur eine Platte als Griff hat eher drückt. Sobald man durch eine Tür hindurch will wendet man automatisch dieses gesammelte Verständnis an um die Tür zu öffnen. An dieser Stelle tritt die Norman-Tür in Kraft. Kann man nicht ausmachen wie oder wo die Tür geöffnet hat scheitert diese Tür an dem simplen und einzigen Anwendungsfall jeder Tür. Der Nutzer sollte nicht erst zweimal überlegen, suchen oder ein Schild lesen müssen um zu wissen wie die Tür funktioniert. Es ist einfach nur eine Tür. Jede Tür die nicht intuitiv richtig geöffnet werden kann ist eine Norman-Tür. Sie führt beim Nutzer zur Verwirrung oder mitunter zur Verzweiflung und Selbstschuld. Doch der Nutzer ist in keinem Fall selbst Schuld. Die Tür legt die Annahme die der Nutzer zur Nutzung trifft durch das Design fest. Die gleiche verwirrte und verzweifelte Reaktion des Nutzer kann auch in der Softwareentwicklung auftreten. Durch die Benutzung der Anwendung bauen Nutzer ein Mentales Modell auf wie sie diese App zu bedienen haben. Daraus entstehen wieder Annahmen und Vermutungen wie bestimmte Anwendungsfälle durchzuführen sind, vergleichbar mit dem öffnen der Tür. Kann der Nutzer also nicht eine Aufgabe auf die Selbe weise wie Checkliste erstellen obwohl sich das Layout gleicht treten beim Nutzer zweifel an seinem Mentalen Modell und Verständnis über die App auf. Dies kann beim

Nutzer zu Verwirrung führen was in einer schlechten Nutzererfahrung mündet. Anwendungen die eine schlechte Nutzererfahrung hervorrufen werden nicht so gern verwendet wie vergleichbare Anwendungen mit guter Nutzererfahrung. Aus diesen Gründen wurde bei der Entwicklung Weert darauf gelegt Anwendungsfälle welche die gleiche Funktion für Checklisten und Aufgaben erfüllen so ähnlich wie möglich zu implementieren. Dies soll zu einer guten Nutzererfahrung und einem einfachen Mentalen Modell zu Benutzung dieser App führen.[4]
Da, wie nun deutlich gemacht wurde, die Implementierung zum Bearbeiten für Checklisten und Aufgaben identisch ist wird der Vorgang nur einmal erläutert. Um Checklisten oder Aufgaben zu bearbeiten wird ein Bearbeitungsmodus eingeführt. Dieser kann durch Betätigen eines Knopfes in der Werkzeugleiste ein und ausgeschaltet werden (vgl. Abbildung 3.1 und Abbildung 3.3). Der Knopf wird durch das Bearbeitungssymbol eines Bleistift dargestellt. Dieser soll klarmachen was dieser Knopf bewirkt. Im nächsten Schritt wird sich die Lösung des Aufgaben speichern Problems zunutze gemacht und um eine Funktion erweitert. Durch tippen auf einen RecyclerView-Listeneintrag mit Aktiviertem Bearbeitungsmodus öffnet sich nun eine Aktivität zum Bearbeiten des Eintrags. Hierbei wird sich auch einer Eigenschaft des Layouts zum erstellen eines Eintrags bedient. Die Text- und Eingabefelder befinden sich nicht fest in dem Layout dieser Aktivität. Sie sind in einer weiteren XML-Datei beinhaltet und werden über den Include-Tag in das Layout dieser Aktivität eingebunden. Dies ist sowohl bei den Checklisten als auch den Aufgaben der Fall. Für das Bearbeiten wurden lediglich neue Aktivitäten mit dem Layout einer Werkzeugleiste, dem einbinden der Text- und Eingabefelder und einem neuen Bestätigungsknopf in der unteren rechten Ecke angelegt. Beim öffnen der Bearbeitungsaktivitäten werden die aktuellen Daten in den Eingabefeldern als Text gesetzt, sodass der Nutzer diese z. B. im Falle eines Tippfehlers einfach beheben kann. Mit bestätigen der Änderungen über den Knopf werden die Daten des gedrückten Elements angepasst. Da die Datei beim speichern komplett neu geschrieben wird sind für das Speichern keine Änderungen nötig. Somit können nun auch die Anwendungsfälle zum Bearbeiten als abgeschlossen angesehen werden.

[4]Interaktive Systeme Vorlesung

3.2.6 Löschen von Checklisten und Aufgaben

Nach dem Bearbeiten soll es dem Nutzer auch möglich gemacht werden Checklisten und Aufgaben zu löschen. Auch bei diesen Anwendungsfällen wird auf die Nutzererfahrung geachtet. Auf diese wurde genauer in Unterabschnitt 3.2.5 eingegangen. Wie beim Bearbeiten auch wird der Anwendungsfall nur einmal beschreiben, da er für Checklisten und Aufgaben identisch ist. Um eine Checkliste Löschen zu können muss der Anwender den Bearbeitungsmodus aktivieren und auf das zu löschende Listenelement tippen. Dadurch öffnet sich die Bearbeitungsaktivität. Wie bereits beschrieben hat diese ebenfalls eine Werkzeugleiste. Dieser wird ein Knopf mit dem Symbol eines Mülleimers hinzugefügt. Nach betätigen des Löschen Knopfes öffnet sich ein Dialogfenster in dem das Löschen des Listenelement bestätigt werden muss. Der Dialog soll das Löschen eines Elements durch versehentliches antippen des Knopfes verhindern. Es wurde jeweils eine Dialog Klasse für Checklisten und Aufgaben angelegt. Diese Klassen erben von der Klasse AppCompatDialogFragment(). Ein Fragment ist ähnlich wie die Aktivität ein Grundlegendes Element von Android. Es handelt sich dabei um einen Teil der Nutzeroberfläche. Fragmente haben ihr eigenes Layout, eigenen Lebenszyklus und kann selbst Nutzereingaben verarbeiten. Fragmente können jedoch nicht eigenständig existieren sondern müssen von einer Aktivität oder einem anderen Fragment gehalten werden. Sie stellen dann einen Teil der Ansicht der Aktivität dar. Um den Dialog zu öffnen wird eine Instanz der erstellten Dialog-Klasse erstellt und über den *.show* Befehl geöffnet. In der Dialog-Klasse wird, wie auch bei Aktivitäten, eine onCreate-Methode überschrieben. Im Gegensatz zu Aktivitäten wird hier nicht das Layout gesetzt sondern in diesem Fall über einen AlertDialogBuilder gebaut. Dem Builder werden Titel, Text und jeweils ein positiver und negativer Knopf mit Text als Komponenten mitgegeben. In der Aktivität die den Dialog erstellt hat wird über eine Methode die Antwort des Nutzers übergeben und die Aktivität im Falle einer Bestätigung beendet. Die Einträge werden gelöscht indem der getippte Listeneintrag aus der jeweiligen Datenliste des Adapter entfernt wird. Identisch zum bearbeiten ist ebenfalls keine Änderung bei der Speichern Funktion notwendig, da beim schreiben der Datei nur die Einträge der Datenliste gespeichert werden.
Mit der Implementation des Löschen sind alle Grundlegenden im Umfang definierten Anwendungsfälle für die erstellten Klassen vorhanden. Der letzte Anwendungsfall, das entfernen aller Haken, stellt eine Zusatzfunktion für eine Bessere Nutzererfahrung dar und wird in Unterabschnitt 3.2.7 erläutert.

3.2.7 All Haken entfernen

Die Idee hinter dieser Anwendung beruht auf dem täglichen Nutzen beim verlassen des Hauses oder Arbeitsplatzes. Je nach Umfang kann eine Checkliste unter Umständen relativ viele Aufgaben beinhalten. Mit dem derzeitigen Funktionsumfang müsste der Nutzer bei jeder erneuten Nutzung einer Checkliste die Haken einzeln von jeder Aufgabe entfernen, was sich als aufwendig herausstellen kann. Aus diesem Grund wurde der Anwendungsfall „Alle Haken entfernen“ implementiert. Für diese Funktion wurde die Werkzeugleiste in der Aufgabenansicht um einen Knopf erweitert. Der Knopf kann bereits auf Abbildung 3.3 links des Bearbeiten-Knopfes gesehen werden. Durch betätigen dieses Knopfes werden alle Checkboxen und die Attribute der Aufgaben, welche des Status halten, auf nicht abgehackt gesetzt.
Mit der Implementation dieses Anwendungsfall sind alle in der Projektdefinition geplanten Anwendungsfälle implementiert und die App hat ihren grundlegenden Status erreicht. Die Funktionalität der einzelnen Anwendungsfälle wird in Abschnitt 3.3 mithilfe von Espresso- und Nutzer-Tests getestet. Die Entwicklung gilt jedoch hiermit als, für diesen Teil, abgeschlossen.

3.2.8 Erweiterung

Wie bereits in Abschnitt 2.3 und Abschnitt 2.4 erläutert sind nach Rücksprache mit dem Betreuer Ideen für weitere Anwendungsfälle aufgetreten. Im Umfang dieser Studienarbeit wurde einer dieser neuen Anwendungsfälle in die Anwendung implementiert. Dabei handelt es sich um den Zeitstempel an den Aufgaben. Dieser soll verdeutlichen ob der Hacken an einer Aufgabe noch von einem Vortag oder doch schon am aktuellen Tag gesetzt wurde. Dazu wurde das Layout des Aufgaben-Listenelements um ein Textfeld für den Zeitstempel erweitert. Das Textfeld wird, in der selben Funktion wie der Haken gesetzt wird, mit dem aktuellen Datum und der Uhrzeit gefüllt. Der Zeitstempel ist in Abbildung 3.3 bereits vorhanden und einsehbar. Die ursprüngliche Position des Zeitstempels war links von der Checkbox. Diese wurde durch Probleme beim Testen auf die jetzige Position verschoben. In Abschnitt 3.3 wird auf diese Positionsänderung weiter eingegangen.
Die beiden anderen Anwendungsfälle haben es Aufgrund von Zeitmangel nicht in den ersten Stand im Rahmen der Studienarbeit geschafft. Hier greif die in der Risikobehandlung

beschriebene Alternative. Somit werden diese Anwendungsfälle aus dem erweiterten Umfang entfernt.

3.3 Tests

Dieser Abschnitt behandelt das Testen der Anwendung. Im Fokus stehen die im Umfang festgelegten Espresso-Tests. Espresso ist eine API zum schreiben von Android User Interface (UI) Tests. Des weiteren wurde die Anwendung im realen Umfeld verwendet und somit einem weiteren Test unterzogen.
Das Ziel von Espresso Tests ist das UI beziehungsweise die UIs durch Simulation von Nutzer Interaktionen zu testen. Damit können die Abläufe von Anwendungsfällen auf ihre Funktionalität getestet werden. Um Espresso zu verwenden müssen dieses zunächst in der Build.gradle Datei importiert werden. Im Anschluss daran kann mit dem schreiben von Espresso Tests begonnen werden. Hierbei ist zu beachten die Test Dateien in der androidTest Ordnerstruktur anzulegen. Zusätzlich muss eine neue „Run Configuration“ angelegt werden. Im Gegensatz zu der Standardmäßig vorhandenen muss hier anstelle von Android App Android Instrumented Test ausgewählt werden. Über diese Konfiguration entscheidet die Entwicklungsumgebung ob die App normal emuliert oder die Tests der androidTest Ordnerstruktur an der App ausgeführt werden sollen. Mit dem Beachten und Einhalten dieser Schritte ist die Einrichtung der Espresso Tests abgeschlossen. Für das Testen der Anwendung wurde die Datei „Espresso Tests“ unter androidTest angelegt. Listing 3.10 zeigt den Beginn der Datei sowie den geschrieben Test für den Anwendungsfall des Checklisten erstellen.

```
1 @RunWith(AndroidJUnit4::class)
2 @FixMethodOrder(MethodSorters.NAME_ASCENDING)
3 @LargeTest
4 class EspressoTests {
5
6     @get:Rule
7     val activityRule = ↙
        ↳ ActivityTestRule(MainActivity::class.java)
8
```

```
 9      @Test
10      fun a_createChecklist() {
11
12          onView(withId(R.id.add_checklist)).perform(click())
13          onView(withId(R.id.titleInput)).perform(typeText("Espresso ↙
                ↳ Checklist Title"))
14          onView(withId(R.id.descriptionInput)).perform(
15              typeText("Espresso Checklist Description"),
16              ViewActions.closeSoftKeyboard()
17          )
18          onView(withId(R.id.createChecklistFab)).perform(click())
19
20          onView(withText("Espresso Checklist ↙
                ↳ Title")).check(matches(isDisplayed()))
21      }
```

Listing 3.10: Start der EspressoTest Datei und Test zum Erstellen der Checklisten

Vor der Deklaration der EspressoTests Klasse wurden Parameter festgelegt welche in dieser Klasse verwendet werden müssen. RunWidth legt fest, dass die Tests innerhalb der Klasse im Stil von JUnit 4 geschrieben werden müssen. Über LargeTest wird dem System mitgeteilt, dass die Laufzeit dieser Tests länger als 1000 Millisekunden ist. Dies ist vor allem Relevant für Klassen die mehrere Tests enthalten oder Tests welche, wie in diesem Fall, die ganze Anwendung Testen. In dieser Test Klasse werden alle Anwendungsfälle der App getestet. Da diese unter Umständen von vorherigen Aktionen abhängig sind wurde der FixMethodOrder Parameter hinzugefügt. Dieser enthält als zusätzliche Info dass die Tests in absteigender Reihenfolge nach dem Namen sortiert ausgeführt werden. Mithilfe dieses Parameters wird sich wiederholender Code in den einzelnen Test Methoden verhindert. Beispielsweise muss eine Checkliste vorhanden sein um sie bearbeiten zu können. Ohne den Zusatz müsste zunächst in der Bearbeiten-Test Methode eine Checkliste erstellt werden obwohl die Erstellen-Test Methode diesen Anwendungsfall bereits testet und eine Checkliste vorhanden wäre. Die Tests werden allerdings nicht in der geschrieben Reihenfolge sondern zur Laufzeit ausgeführt. Daher könnte es vorkommen, dass zuerst die Bearbeiten Methode ausgeführt wird und diese sonst aufgrund einer fehlenden Checkliste fehlschlagen würde. Um den redundanten Code zu vermeiden wurden die Test Methoden so benannt, dass diese in der gewollten Reihenfolge ausgeführt werden und die bearbeiten

Methode so nicht zuerst selbst eine Checkliste erstellen muss.
Innerhalb der Klasse wird noch mit dem Zusatz @Rule die Startaktivität MainActivity als Einstiegspunkt für die Tests festgelegt. Jede Methode welche einen Test repräsentieren soll muss mit @Test als Test deklariert werden. Innerhalb einer Test Methode werden die Normalerweise vom Nutzer getätigten Interaktionen geschrieben und am Ende das Ergebnis mit dem erwarteten Ergebnis verglichen, um so den Anwendungsfall zu Testen. Espresso bedient sich hierbei den IDs der Elemente auf dem Layout. So wird beispielsweise der Knopf zum hinzufügen einer Checkliste gefunden und mit der Espresso Methode perform(click()) das Tippen auf den Knopf simuliert. Auf gleiche weise funktioniert auch das Eingeben von Texten, allerdings wird hier typeText("Text") anstelle von click() verwendet. Bei der Eingabe von Text ist zu beachten dass nach der Eingabe die Bildschirmtastatur geschlossen werden muss, da diese sich durch die typeText-Methode öffnet und andere Elemente auf dem Bildschirm verdecken kann. Über die check(Matches(isDisplayed())) wird in diesem Fall überprüft ob nach dem Erstellen der Checkliste ein Element mit dem entsprechenden Text auf dem Bildschirm angezeigt wird.
Die Tests der restlichen Anwendungsfälle folgen dem gleichen Schema. Die RecyclerViews und das damit verbundene Klicken auf ein bestimmtes Element hat während dem Schreiben von Testen für Probleme gesorgt, da diese eine Eigene Ansicht darstellen und die einzelnen Listenelemente nicht in der MainActivity gefunden werden können. Das Problem konnte jedoch behoben und alle Tests erfolgreich geschrieben werden. Die Lösung des Problems wird in Abschnitt 3.4 behandelt.
Nach erstem Testen der Anwendung mithilfe der Espresso Test wurde nach der Rücksprache mit dem Betreuer eine erste Version der Anwendung erstellt, um die App durch wirkliche Nutzung zu testen. Hierfür haben mehrere Testpersonen die Anwendung auf ihrem Smartphone installiert. Die Testpersonen bestanden unter anderem aus Familienmitgliedern. Von einer der Testpersonen kam das Feedback den Zeitstempel zu verschieben, da dieser bei bestimmter Länge des Aufgabentitels von diesem Überdeckt wird. Die App wurde über den Zeitraum der Klausurphase (zwei Wochen) verwendet und getestet. Während dieser Zeit sind bei der Nutzung keine weiteren Fehler oder Probleme aufgetreten. Die App hat ihre Intention mehr als Zufriedenstellend erfüllt und mehrere doppelte Wege erspart.
Das Testen der App kann im Umfang der Studienarbeit als erfolgreich angesehen werden. Durch die zusätzlichen Nutzungstests wurde der definierte Testumfang sogar übertroffen.

3.4 Problembehandlung