CharBuilder
Mobile Applikationen

Fachhochschule Bielefeld
Campus Minden
Studiengang Informatik

Beteiligte Personen:

Name	Matrikelnummer
Christopher Kluck	1078455
Philipp Clausing	1078231

---

22. Januar 2018

Inhaltsverzeichnis

1 Einleitung 4

2 Stand der Technik 5
- 2.1 Android . . . 5
- 2.2 Kotlin . . . 5
- 2.3 Gson . . . 6
- 2.4 Gradle . . . 6
- 2.5 Android Studio . . . 6
- 2.6 Git . . . 7
- 2.7 UMLet . . . 7
- 2.8 Google Firebase . . . 7

3 Anforderungen 8
- 3.1 Must Have . . . 8
- 3.2 Should Have . . . 9
- 3.3 Could Have . . . 9
- 3.4 Won't Have . . . 9

4 Architektur 10
- 4.1 Klassendiagramm . . . 10
  - com.chrisphil.charbuilder . . . 10
  - com.chrisphil.charbuilder.controller . . . 11
  - com.chrisphil.charbuilder.creationfragments . . . 11
  - com.chrisphil.charbuilder.help . . . 11
- 4.2 Model View Controller(MVC) . . . 12

5 Implementierung 13
- 5.1 Charaktererstellung . . . 13
  - ObligationFragment.kt . . . 14
  - SpeciesFragment.kt . . . 15
  - CareerFragment.kt . . . 16
  - SpecializationFragment.kt . . . 16
  - ExperienceFragment.kt . . . 16
  - AttributeFragment.kt . . . 17
  - MotivationFragment.kt . . . 17
  - AppearanceFragment.kt . . . 19
  - GearFragment.kt . . . 19
- 5.2 Würfeltool . . . 20
- 5.3 Import/Export . . . 21

6 Test und Usability 21

7 Zusammenfassung 22
7.1 Fazit 22
Christopher Kluck 22
Philipp Clausing 23
7.2 Entwicklungspotenzial 23

1 Einleitung

In diesem Dokument wird die Entwicklung der Applikation CharBuilder als Semesterprojekt für das Wahlpflichtfach Mobile Applikationen im fünften Semester dargestellt. Die Aufgabenstellung hat uns sowohl die Wahl der Plattform als auch die Art der App offengelassen. Da wir in der Vorlesung die Grundlagen der App-Entwicklung unter Android gelernt haben, machte es für uns Sinn, unsere erste App für Android zu entwickeln. Weitere Faktoren für die Entscheidung waren, dass man für die Entwicklung keinen teuren Development Account benötigt und beide Entwickler bereits Kenntnisse in Java besitzen. Außerdem kann die App direkt auf dem eigenen Smartphone getestet werden.

Die Idee der App entstand aus unserem gemeinsamen Hobby, dem Pen and Paper Rollenspiel. Pen and Paper Charaktere werden normalersweise am Computer generiert und verwaltet. Wenn man sich zum Spielen trifft, bringt jeder seinen Charakterbogen ausgedruckt oder als Datei auf dem Handy mit. Ziel der App ist es, sowohl neuen als auch erfahrenen Pen and Paper Spielern eine komfortable Lösung zu Erstellung und Verwaltung ihrer Charaktere auf dem Handy zu bieten. Dadurch kann man sich das Ausdrucken oder Übertragen aufs Handy sparen und man kann seine Charaktere überall mit hinnehmen.

Der Aufgabenstellung konnten wir die fünf Themengebiete Stand der Technik, Anforderungen, Architektur, Implementierung und Test und Usability für unsere Dokumentation entnehmen. Auf jedes dieser Themen werden wir im Verlauf der Dokumentation eingehen und dadurch strukturiert die Entwicklung der App aufzeigen.

2 Stand der Technik

Im Nachfolgenden wird erläutert, welche Technologien wir für die Erstellung der CharBuilder App verwendeten und warum diese von uns gewählt wurden. Hierbei haben wir den Text in die Unterkategorien Android(Betriebssystem), Kotlin(Programmiersprache), Gson(Json Java-Lib), Gradle(Build Tool), Android Studio(IDEA), UMLet(UML Tool), Git und Google Firebase aufgeteilt. Das sind die wichtigsten Technologien, die wir zum Entwickeln unserer App genutzt haben.

2.1 Android

Wie in der Einleitung geschrieben, haben wir uns für Android als Betriebssystem entschieden. Da verschiedene Version dessen auf unterschiedlichen Geräten verteilt sind muss man sich als Entwickler darüber hinaus auch für ein Mindest-API-Level entscheiden. Dies stellt die niedrigste Android Version dar, unter welcher die App ausgeführt werden kann. Bei der Wahl des API-Levels spielen verschiedene Faktoren eine Rolle, unter anderem wie viele der Android Geräte welche Version des Betriebssystems ausführen oder ob der Entwickler Funktionen nutzen möchte die erst ab einer bestimmten Version verfügbar sind.
Wir haben uns entschieden, unsere Applikation lediglich für Android Betriebssysteme der Version 5.0(API Level 21) oder höher zu entwickeln. Die Version 5 ist bereits am 12.11.2014 veröffentlicht worden und bringt einen großen Wandel in der Benutzeroberfläche durch Googles Material Design. Dieses ist an den Gestaltungsstil "Flat Designängelehnt und minimalistisch gehalten. Die Entscheidung trafen wir aufgrund der weitreichenden Änderungen in dieser Version, des bereits 3 Jahre in der Vergangenheit liegenden Veröffentlichungsdatums und der breiten Verteilung von 80,7% aller Android Geräte.

2.2 Kotlin

Kotlin ist eine neue Programmiersprache von JetBrains aus dem Jahr 2016. Wie auch Java kompiliert Kotlin zu JVM Bytecode. Es lässt sich dadurch sehr gut in das bestehende Java Ökosystem einbinden und kann alle Java Bibliotheken verwenden. Seit dem 17. Mai 2017 ist Kotlin eine von Android offiziel unterstützte Sprache.[Kaz17]

Beide Teammitglieder haben zuvor noch nicht mit Kotlin gearbeitet, konnten sich jedoch aufgrund der Ähnlichkeit zu Java schnell zurechtfinden und gute Erfahrungen sammeln. Die Programmiersprache bietet dem Programmierer viele Vorteile wodurch sie immer beliebter wird unter Androidentwicklern. Der weitreichenste Vorteil ist es null-Referenzen zu verhindern, die Sprache bietet einem "Null Safety". Eine weitere Änderung ist auch die Möglichkeit Datenklassen zu erstellen, diese sind Klassen welche lediglich Daten halten und keinerlei Methoden selbst implementieren. Ein

Beispiel dieser kann man im Kapitel Implementierung finden. Der Großteil der Appentwicklung geschah mit Kotlin v1.15 später wurde dann ein Update auf Kotlin v1.2 durchgeführt. Dies liegt an dem Entwicklungszyklus von Kotlin, so wurde immer mit dem aktuellsten stabilen Release gearbeitet.[kot16]

2.3 Gson

Gson ist eine Java Bibliothek zur Serialization und Deserialization. Sie wird genutzt um Java Objekte in JSON umzuwandeln oder auch JSON zu Java Objekten zu wandeln. Hierbei ist die Möglichkeit Generics zu verwenden äußerst wichtig, da diese in der CharBuilder Applikation mehrmals zum Einsatz kommt. Wir haben diese Bibliothek ausgesucht da sie uns bereits bekannt war von früheren Projekten und wir positive Erfahrungen gemacht haben. Desweiteren wird die Software bereits seit 2008 entwickelt und konnte seitdem durch viele Revisionen und Verbesserungen überzeugen. Für die App verwendet wir die Gson Version 2.8.2.[Goo13]

2.4 Gradle

Gradle ist ein weit verbreitetes Build-Tool, welches automatisiert arbeitet und Unterstützung für mehrere verschiedene Sprachen bietet. Es wird typischerweise für Android Applikationen verwendet, welche mit Android Studio entwickelt werden, da das Tool tief in die Entwicklungsumgebung integriert ist. Zu unserer Entscheidung dieses Programm zu verwenden kann nicht viel gesagt werden, da es wie oben erwähnt bereits integriert war und eine Alternative neben großem Mehraufwand keine Vorzüge geboten hätte.

2.5 Android Studio

Android Studio ist die offizielle Entwicklungsumgebung für native Androidprogrammierung. Es bietet neben den bekannten IDE Funktionen wie Syntax Highlighting, Autovervollständigung, Instant Run(Es wird nur der veränderte Teil neu kompiliert) und Debugger auch einen Android Emulator, um verschiede Geräte und Android Version zu simulieren. Die IDE basiert auf IntelliJ's Softwareprodukten. Mit der Version 3.0.1 wurde die Entwicklungsumgebung für Kotlin angepasst, dies ist auch die Version welche wir zur Entwicklung unserer App verwendeten. Dabei liefert die neue Version ebenfalls ein Programm, um Java-Code direkt in Kotlin-Code umzuwandeln. Dem Entwickler wird so der Einstieg in Kotlin erleichtert, das Tool sollte im späteren Verlauf aber nur selten genutzt werden, um eine einheitliche Codequalität zu erreichen.

2.6 Git

Git ist eine Software zur Versionsverwaltung. Es findet bereits große Anwendung in der Industrie und ist neben SVN das bekannteste System. Hauptgrund für unsere Entscheidung Git zu nutzen war die bereits über die vergangene Semester gesammelte Erfahrung. Git bietet weiterhin den Vorteil das man getrennt arbeiten kann, da jeder eine lokale Arbeitskopie hat. Für unser Projekt verwendeten wir den Git-Serviceprovider Github, dieser stellt kostenloses ein Repository für uns zu Verfügung.

2.7 UMLet

UMLet ist ein kostenloses Open-Source UML Tool mit einer simplen Benutzeroberfläche. Die erstellten Diagramme lassen sich z.B. als pdf, jpg oder png exportieren. Wenn einem die große Auswahl an UML Elementen nicht reicht, kann man eigene erstellen oder importieren. Zum Zeichnen unserer Diagramme haben wir die Version 14.2 genutzt.[Aue]

2.8 Google Firebase

Firebase ist eine Entwicklungsplattform für Mobile- und auch Webapplikationen. Ursprünglich als Startup 2011 gegründet wurde es 2014 von Google aufgekauft. Der Dienst bietet mehrere Funktionen, in unserer App verwenden wir allerdings nur *Firebase Storage* verwendet um dem Nutzer das speichern seines Charakters zu ermöglichen. Hier bietet der Service eine Stufe welche es uns ermöglicht die Anwendung im kleinen Rahmen zu verwenden ohne das zusätzliche Kosten entstehen.

3 Anforderungen

Im folgenden Kapitel werden die Anforderungen, die wir an unsere App gestellt haben, genauer beschrieben. Diese Anforderungen wurden von uns vor Beginn der App-Implementierung aufgestellt, um einen Überblick über alle Funktionen zu erhalten, die die App beinhalten wird. Die Anforderungen sind in diese vier Kategorien eingeteilt:

Must Have
: Unter "Must have" ist jede Funktionalität aufgelistet welche **unbedingt** umgesetzt werden muss, um das Projekt als erfolgreich bezeichnen zu können. Diese Kategorie wird oft auch "Minimale Anforderungen" genannt.

Should Have
: Die Kategorie "Should have" fasst all die Punkte unter sich zusammen, die für ein erfolgreiches Projekt nicht unbedingt erforderlich sind, aber dennoch wichtiger Bestandteil sein können. Als Beispiel sei eine Funktionalität zu sehen, welche das Produkt lediglich erweitert, von welcher die Grundfunktionalitäten allerdings nicht abhängig sind.

Could Have
: Features, die unter "Could have" aufgelistet sind, werden nur umgesetzt, wenn alle Punkte unter "Must have" und "Should have" bereits abgearbeitet sind. Sie sind vollkommen optional.

Won't Have
: "Won't have" beinhaltet jene Punkte, welche **nicht** umgesetzt werden. Sie wurden von den Entwicklern oder Auftraggebern zu Beginn der Projektspezifikation ausgeschlossen.

3.1 Must Have

Als wichtigste Funktion der App gilt das Erstellen eines neuen Charakters. Es muss möglich sein einen Charakter entsprechend des gewählten Regelwerks zu erstellen. Zunächst wird es nur möglich sein Charaktere im Regelwerk *Star Wars: Am Rande des Imperiums* zu erschaffen. Während des Erstellungsprozesses kann der Nutzer auf Regeltexte zurückgreifen, die ihm seine aktuellen Auswahlmöglichkeiten erklären. Diese Regeltexte werden aufgrund von eventuellen Urheberrechtsverletzungen nicht aus dem offiziellen Regelwerk kopiert, sondern von den Entwicklern der App zusammengefasst.

Nachdem der Nutzer einen Charakter erstellt hat, wird dieser in einer Liste mit anderen bereits von ihm angelegten Charakteren angezeigt. In der Liste wird das für den Charakter ausgewählte Bild, sein Name und das Regelwerk, in dem er

gespielt wird, dargestellt. Der Nutzer kann nicht mehr benötigte Einträge aus der Liste löschen, Änderungen an den Charakteren vornehmen und diese als PDF-Datei exportieren. Durch einen Klick auf einen der Listeneinträge werden dem Nutzer alle Eigenschaften, Talente, Spezialisierungen, etc. zu einem Charakter angezeigt.

Eine weitere Funktion der App ist das eingebaute Würfeltool. Wenn der Nutzer sich in der Detailansicht zu einem seiner Charaktere befindet, kann er ohne großen Aufwand Proben auf seine Fähigkeiten werfen. Es wird dann berechnet, ob die Probe erfolgreich war oder nicht und das Ergebnis erscheint in einem Dialog. Außerdem wird das Ergebnis des Würfelwurfs im Verlauf des Würfeltools angezeigt, sodass der Nutzer eine Übersicht seiner vorherigen Proben erhält. Das Würfeltool ist über die Navigationsleiste im Hauptmenü erreichbar und bietet eine Auswahl an Würfeln, die für das jeweilige Regelwerk benötigt werden.

3.2 Should Have

Damit Charaktere nicht an ein bestimmtes Gerät gebunden sind, sollte der Nutzer die Möglichkeit haben, seine Charakterdaten als JSON-Datei in Google Drive zu speichern. Jeder Charakter bereist im Zuge seiner erlebten Abenteuer eine Vielzahl von Orten. Um dem Spieler eine Erinnerungsstütze zu geben, an welchen Orten sein Charakter schon war und was für Abenteuer er dabei abgeschlossen hat, sollte es eine Art Reisetagebuch für jeden Charakter geben.

3.3 Could Have

Es wird nicht ausgeschlossen, dass die App die Erstellung von Charakteren in unterschiedlichen Regelwerken ermöglicht. Allerdings gibt es große Unterschiede in der Komplexität und dem Umfang von Pen and Paper Regelwerken, weswegen zunächst nur ein Regelwerk unterstützt wird. Eine englischsprachige Version der App ist geplant. Das Übersetzen der Regeln nimmt jedoch viel Zeit in Anspruch. Eine weitere mögliche Option wäre das Einbinden der Google Billing API, die dem Entwickler das Freischalten von In-App Käufen ermöglicht.

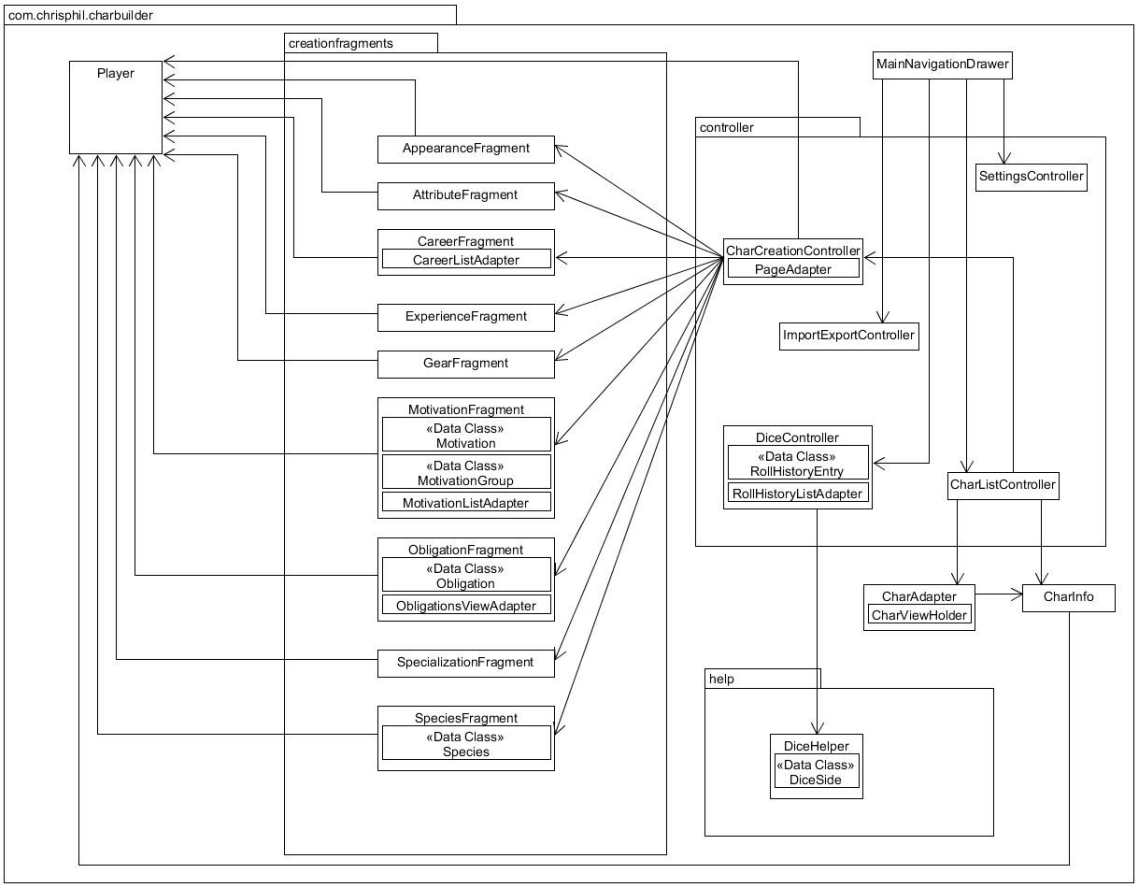
3.4 Won't Have

Die App wird keine Gruppenverwaltung ermöglichen. Das bedeutet, dass ein Spielleiter nicht die Charaktere seiner Gruppenmitglieder einsehen kann. Außerdem werden in der App keine spielbaren Abenteuer vorhanden sein. Sie dient lediglich dem Speichern und Verwalten von Pen and Paper Charakteren. Nutzer der App werden in der App keine Anleitung für das Spielen von Pen and Paper Abenteuern finden. Alle verwendeten Auszüge aus Regelwerken dienen ausschließlich der Unterstützung während des Erstellungsprozesses.

4 Architektur

Dieses Kapitel beschäftigt sich mit der Architektur unserer App. Dazu werden wir zunächst den Aufbau der Packages und die Beziehungen der einzelnen Klassen untereinander anhand des Klassendiagramms darstellen, bevor wir näher auf das Model View Controller Pattern eingehen.

4.1 Klassendiagramm



Das oben zusehende Klassendiagramm zeigt alle für die Applikation gebrauchten Klassen und die Pakete, in die sie eingeteilt sind. In den nächsten Absätzen werden die einzelnen Pakete und ihre zugehörigen Klassen bescreiben.

com.chrisphil.charbuilder

Eine Einordnung in eindeutige Pakete ist nicht für jede Klasse möglich. Aus diesem Grund werden solche Klassen in diesem Paket zusammengefasst.

Der *MainNavigationDrawer* ist eine dieser Klassen. Als Mainactivity der App können von ihm die Klassen *CharListController*, *DiceController*, *SettingsController* und *ImportExportController* aufgerufen werden.

Als meistgenutzte Klasse hat *Player* die meisten Verbindungen zu anderen Klassen. Das Laden und Speichern der Charakterdaten wird in dieser Klasse umgesetzt. Außerdem werden Daten aus den "CreationFragments" in einer Instanz der *Player*-Klasse während der Charaktererstellung zwischengespeichert.

Die letzten beiden Klassen dieses Pakets sind der *CharAdapter* und die *CharInfo*. Der *CharAdapter* dient als Adapter, der für die Charakterliste des *CharListControllers* eingesetzt wird. Den Inhalt dieser Liste bildet die *CharInfo*, welche die geladenen Daten aus der *Player*-Klasse mit den zugehörigen Bildern der Charaktere verbindet und an den *CharAdapter* weitergibt.

com.chrisphil.charbuilder.controller

Unsere ursprüngliche Architektur sah vor, dass jedes Fragment, das durch den Nutzer über den *MainNavigationDrawer* erreicht werden kann, einen eigenen Controller besitzen wird. Auf diese Weise sollten Darstellung und Datenverarbeitung getrennt werden. Alle benötigten Controller werden in diesem Paket gebündelt. Da diese Architektur sich als unpraktisch erwiesen hat, wurden alle mit dem *MainNavigationDrawer* verbundenen Klassen zu eigenständigen Fragmenten ohne zugehörigen Controller.

Die einzige Klasse, welche ihre Funktion als Controller mehr oder weniger behalten hat, ist der *CharCreationController*. Sämtliche Navigation der Charaktererstellung wird durch diese Activity geleitet, weshalb sie eine Verbindung zu jedem der "CreationFragments" besitzt. Außerdem wird in ihr eine Instanz der *Player*-Klasse erzeugt.

com.chrisphil.charbuilder.creationfragments

Jedes der Fragmente in diesem Paket wird während der Charaktererstellung benötigt. Einige verfügen über innere Adapter-Klassen zum Füllen von Listen. Fragmente mit "Data Classes" lesen Daten aus XML-Dateien ein. Die Daten der Fragmente werden während der Charaktererstellung in einem Objekt der *Player*-Klasse zwischengespeichert, welche zur *CharCreationController*-Activity gehört.

com.chrisphil.charbuilder.help

Das Paket "Help" wurde erstellt, um Klassen zu sammeln, die eventuell mehrfach genutzte Programmlogik enthalten. So kann der Code aus dem *DiceHelper* z.B. nicht nur im *DiceController* angewendet werden, sondern bei Bedarf auch aus anderen Klassen heraus aufgerufen werden.

4.2 Model View Controller(MVC)

Model View Controller[Mun] ist ein bekanntest Software Architektur *Pattern*. In diesem wird der Quellcode in die 3 Bereiche Model(Datenrepräsentation), View(Darstellung) und Controller(Programmlogik) aufgeteilt. Das dient der Wiederverwendbarkeit von Code und unterstützt durch Kapselung die parallele Programmierung.
Das Anwenden des MVC-Patterns ist nicht unüblich bei Android Applikationen, so entschieden wir uns auch dafür. Ursprünglich war geplant das die jeweiligen Controllerklassen (*CharCreationController*, *DiceController* usw.) die Programmlogik darstellen und neben den Views(Layoutdateien) die Playerklasse das Model repräsentiert. Dies lies sich aufgrund der vielen Fragmente bei der Charaktererstellung nur bedingt einhalten, das parallele Programmieren wurde durch diese Struktur dennoch unterstützt.
Die Klasse *Player* welche wie erwähnt als Model dient wurde von uns ebenfalls entwickelt damit alle Fragmente zu jeder Zeit die aktuellen Daten einsehen und so auf Veränderungen reagieren können. Insgesamt hatte das *Pattern* einen positiven Einfluss auf unsere Softwarearchitektur. Um MVC umzusetzen mussten wir uns im Vorfeld bereits mit Struktur, Datenkapselung und Modellen beschäftigen, so dass von beginn an der Aufbau der Applikation festgelegt war.

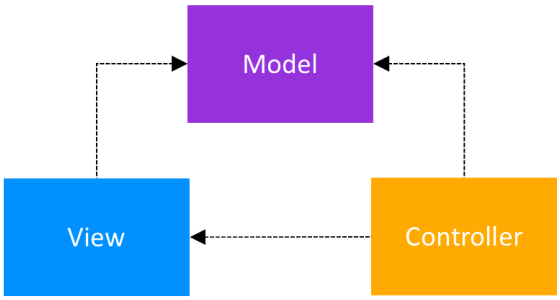


Abbildung 1: Das Model View Controller Pattern

5 Implementierung

5.1 Charaktererstellung

Da die Erstellung eines Charakters im offiziellen Regelwerk bereits in sinnvolle Schritte unterteilt war, wollten wir diese Gliederung in der App wiederspiegeln. Durch das Bewahren der in den Regeln eingeführten Struktur haben neue Spieler einen guten Überblick über die für die Charaktererstellung benötigten Schritte. Außerdem müssen sich Spieler, die das offizielle Regelwerk bereits kennen, nicht an eine neue Gliederung gewöhnen. In der App werden die Schritte durch Fragmente dargestellt. Jedes Fragment besitzt ein eigenes Layout, das an die Anforderungen des jeweiligen Erstellungsschritts angepasst ist. Für die Navigation zwischen den einzelnen Fragmenten verwenden wir ein *TabLayout*. Dieses Layout wird durch die Klasse *CharCreationController* mit den nötigen Tabs versehen.

Der *CharCreationController* ist eine Activity, die geöffnet wird, sobald der Nutzer die Charaktererstellung startet. Er besitzt eine innere Klasse namens *PageAdapter*, welcher von der durch Android zu Verfügung gestellten Klasse *FragmentStatePagerAdapter* erbt. Durch den *PageAdapter* wird geregelt, wann ein neues Fragment geladen wird und an welcher Stelle des *TabLayouts* dieses Fragment eingeordnet wird. Das *TabLayout* bietet dem Nutzer eine gute Übersicht über die Schritte der Charaktererstellung. Durch Klicken auf die Namen der einzelnen Tabs oder das Wischen nach links oder rechts kann der Nutzer frei wählen, welche Schritte des Erstellungsprozesses er bearbeiten möchte. Zudem kann er nachträglich Änderungen an Fragmenten vornehmen, falls ihm seine derzeitige Wahl nicht gefällt.

Jede Eingabe, die der Nutzer während der Erstellung in den Fragmenten einträgt, wird an die Activity weitergegeben, welche die Daten in einer Instanz der Klasse *Player* speichert. Bei Initialisierung eines neuen Fragments wird geprüft, ob schon Werte in den entsprechenden Attributen des Player-Objekts vorhanden sind. Hat der Nutzer das geladene Fragment schonmal bearbeitet und möchte jetzt Änderungen vornehmen, sollen die Felder wieder so ausgefüllt sein, wie er sie hinterlassen hat. Eine weitere potenzielle Quelle für Verlust von Daten ist das versehentliche Drücken der Zurück-Taste auf der *Android Navigation Bar*. Zur Vermeidung dieses Problems fängt die Activity Klicks auf die Zurück-Taste ab und zeigt einen Dialog an, in dem man das Abbrechen der Charaktererstellung bestätigen muss.

Es gibt zwei Layouts, die in mehreren Fragmenten als Listenelement in *ListViews* oder als Group Item in *ExpandableListViews* eingesetzt werden:

char_creation_listview.xml
: Ein Layout bestehend aus zwei *TextViews* für Name und kurz Beschreibung des jeweiligen Elements und einem anklickbaren *ImageView*, das, wenn der

Nutzer darauf drückt, einen Dialog mit dem Regeltext zum jeweiligen Element wiedergibt.

char_creation_career_group_item.xml
: Dieses Layout besteht nur aus einem *TextView* und wird in *ExpandableListViews* als Group Item Layout verwendet.

Einige der Fragmente verfügen über eine *Data Class*. Diese Klassen sind ein in Kotlin eingeführtes Konzept, um das Schreiben von unnötigem "Boilerplate Code" in einfachen Klassen, die nur als Datencontainer genutzt werden, zu verhindern [Par17]. Häufig werden die Datenklassen in den Fragmenten dazu genutzt, Daten aus einer geparsten XML-Datei zu speichern und sie dann in ein *ListView*-Element zu laden. Mit Hilfe dieser Methode erleichtern wir uns das Zuordnen der passenden Kurzbeschreibungen und Regeltexte zu einem entsprechenden Regelbegriff.

Im Folgenden werden wir auf die einzelnen Fragmente und ihre Funktionen eingehen. Sie sind in der Reihenfolge angeordnet, in der sie auch in den Tabs der Charaktererstellung wiederzufinden sind.

ObligationFragment.kt

Der erste Schritt in einer Charaktererstellung ist das Bestimmen der Verpflichtung/en eines Charakters. Das Regelwerk zu „Star Wars : Am Rande des Imperiums" [Her13, 39] gibt eine Liste von zwölf wählbaren Pflichten vor. Dem Spieler werden zwei Möglichkeiten angeboten, eine Pflicht für seinen Charakter zu wählen:

1. Der Spieler sucht sich aus der Liste mit zwölf Einträgen eine für ihn passende Pflicht aus.

2. Durch das Drücken auf einen Button lässt der Spieler die App per Zufall ein oder zwei Pflichten bestimmen.

Zur Umsetzung der Liste haben wir uns für eine *ListView* entschieden, die durch die innere Klasse *ObligationsViewAdapter* mit Inhalt gefüllt wird. Der *ObligationsViewAdapter* ist eine Unterklasse des *BaseAdapters* aus der Android API, welcher in der *onCreateView*-Methode des Fragments als Adapter der *ListView* gesetzt wird. Mit der Liste aus Pflichten, die zuvor aus einer XML-Datei geparst wurde, erstellt der Adapter dann jedes Listenelement anhand des Layouts aus der *char_creation_listview.xml*. Nachdem ein Listenelement angeklickt wurde, wird es blau hinterlegt und der Name der Pflicht wird als aktuell ausgewählte Pflicht im oberen Textfeld angezeigt.

Jede Pflicht hat eine Wahrscheinlichkeit von 8% durch den Zufall ausgewählt zu werden. Insgesamt hat der Spieler also eine Chance von 96% nur eine Pflicht bei der zufälligen Bestimmung zu erhalten. Die übrigen 4% bilden die Wahrscheinlichkeit dafür, dass der Charakter mit zwei Pflichten startet. Diese Warscheinlichkeiten für

das Auswählen der Pflichten wurden dem Regelwerk [Her13, 39] entnommen, das eine Tabelle mit Würfelergebnissen für jede Verpflichtung beinhaltet. Während der Erstellung ist die Zufallsbestimmung der einzige Weg, eine zweite Verpflichtung zu erhalten. Aus diesem Grund musste in der Implementierung darauf geachtet werden, dass das Textfeld der zweiten Pflicht zurückgesetzt wird, wenn eine neue ausgewählt wird. Dabei ist es egal, ob die neue Verpflichtung durch den Spieler oder den Zufall gesetzt wird. Wenn die derzeitige Pflicht dem Spieler nicht gefällt, kann er sie beliebig oft selbst ändern oder durch Drücken des Buttons neu bestimmen. Die Reihenfolge der Aktionen spielt dabei keine Rolle.

SpeciesFragment.kt

Das SpeciesFragment bietet dem Nutzer eine Liste an, aus welcher ein Element ausgewählt werden muss. Zusätzlich bietet jedes Listenelement eine weitere Schaltfläche an, welche ein Dialogfenster mit zusätzlichen Informationen öffnet. Zu Beginn der Implementierung haben wir uns überlegt wie man die einzelnen Spezies gut darstellen kann ohne den Anwender mit den vielen Informationen zu überfordern. Hier entschieden wir uns dann für die oben genannte Liste, welche jeweils den Namen und eine beschreibenden Satz zu jeder Spezies anzeigt. Sollte der Benutzer sich dann noch weitere Informationen anzeigen lassen wollen so kann er das mit der auf der linken Seite dargestellten Schaltfläche tun. Intern wird jedes Element durch die Klasse *Species* repräsentiert. Diese enthält insgesamt 13 Variablen um jede Eigenschaft gut darstellen zu können. Ein wichtiger Punkt war für uns das speichern der Daten. Hier entschieden wir uns für eine XML-Datei, da Android bereits Funktionen zum Laden und Manipulieren dieser bietet. Wir legten also zusätzlich zum *SpeciesFragment* und der entsprechenden Layout-Datei auch eine *Species.xml* an in welche wir die statischen Daten eingetragen haben. Diese wird dann mittels des *XMLPullParser*'s eingelesen und die entsprechenden Spezies Objekte erstellt. Das Layout des Fragmentes ist entsprechend simpel aufgebaut und enthält lediglich eine *ListView*. Mittels der von uns angelegten Klasse *SpeciesViewAdapter* konnten wir den Inhalt der Listenelemente dann entsprechend setzen. Zuletzt erstellten wir dann eine Methode *createSpeciesInfoDialog* welche ein Spezies Objekt erwartet und aus diesem dann einen entsprechenden Dialog erzeugt welcher mehr Informationen darstellt, da in der Liste lediglich Name und Beschreibung angezeigt werden. Die Liste befindet sich im sogenannten *SINGLE_CHOICE_MODE* damit der Nutzer wirklich nur ein Element auswählen kann, dieses wird dann farblich hervorgehoben. Zu diesem Fragment lässt sich abschließend sagen, dass die Implementierung beim ersten Ansatz gut umgesetzt werden konnte. Dies liegt unter anderem an dem einfachen Aufbau und der guten XML-Datei Unterstützung der Android API.

CareerFragment.kt

Das CareerFragment biete eine ähnliche Funktionalität wie das oben genannten SpeciesFragment, es ist jedoch aufgrund der gesteigerten Komplexität nicht so unkompliziert zu implementieren. Wegen der größeren Menge an Daten und Abhängigkeiten, welche dargestellt werden müssen, haben wir uns entschieden als Hauptelement der Benutzeroberfläche eine *ExpandableListView* zu verwenden. Sie besteht aus *GROUP_ITEM*'s welche jeweils beliebig viele *CHILD_ITEM* haben können. Diese *CHILD_ITEM*'s werden bei der Auswahl des Gruppenelements "ausgeklappt". Auf diese weise ist es möglich die Abhängigkeiten für den Nutzer gut sichtbar zu modellieren. Sollte der Anwender dann eines der Subelemente auswählen, öffnet sich der entsprechende *Skill Tree* und die Applikation merkt sich welcher Beruf ausgewählt wurde. Da die Daten dieses Fragmentes an die Auswahl der Spezies gebunden ist, wird der Nutzer bei Änderungen oder Inkonsistenzen mittels eines Dialoges benachrichtigt. Durch die vielen Elemente mussten bereits nur für die Liste drei Layout-Dateien angelegt werden. In der *char_creation_career.xml* wird lediglich die *ExpandableListView* angelegt, die anderen beiden Dateien sind jeweils für das Gruppen- und Kindelement: *char_creation_career_group_item.xml* und *char_creation_career_child_item.xml*. Im eigentlichen Quellcode ähnelt dieses Fragment sehr dem oben genannten, ein möglicher Beruf wird um Beispiel durch die Klasse Career repräsentiert. Sie bietet unter anderem die Variablen für Namen und die entsprechenden Unterklassen welche die Kindelemente darstellen. Die Daten werden wieder mit dem *XMLPullParser* eingelesen, womit dann die Objekte erstellt werden. Die Daten zu den *Skill Tree*'s werden ebenfalls aus einer XML-Datei geladen. Sollte der Nutzer seine bisherige Auswahl ändern wollen so kann er dies mit einem einfachen Button tun.

SpecializationFragment.kt

Als wir anfangs den Aufbau unserer App geplant haben wurde festgelegt das jeder Schritt des Regelwerks als ein Fragment der Charaktererstellung umgesetzt wird. Als wir allerdings das *CareerFragment* implementiert haben, wurden wir auf die enge Verzahnung der beiden Schritte aufmerksam. Aufgrund dieser Erkenntnisse entschieden wir, dieses Fragment komplett von dem Inhalt des *CareerFragment*'s abhängig zu machen. Sobald sich die Daten in diesem ändern, wird das *SpecializationFragment* zurückgesetzt. So kann gewährleistet werden das der Nutzer keine Eingaben tätigt welche sich gegenseitig widersprechen.

ExperienceFragment.kt

Nachdem der Spieler sich im vorherigen Fragment eine Spezialisierung seines Berufs ausgesucht hat, kann er hier Erfahrungspunkte einsetzen, um bestimmte Fähigkeiten zu erlernen. Dazu wird ihm der Talentbaum seiner Spezialisierung und die einsetzbaren Erfahrungspunkte angezeigt. Wie viele Erfahrungspunkte dem Spieler zum

Erlernen neuer Fähigkeiten zur Verfügung stehen, hängt von der zuvor gewählten Spezies ab. Das bedeutet, das beim Öffnen des Fragments einige Voraussetzungen geprüft werden müssen. Sind diese Bedingungen nicht erfüllt, wird dem Nutzer ein Dialog angezeigt, in welchem er über die Änderungen im Fragment informiert wird.

Ein möglicher Auslöser für einen solchen Dialog ist das Wechseln der Spezies durch den Spieler. Wird dem Charakter eine neue Spezies zugewiesen, ändert sich dadurch die Menge an Erfahrungspunkten, die der Spieler im *ExperienceFragment* ausgeben kann. Da der Spieler durch die Änderung der Spezies eventuell nicht mehr genügend Erfahrungspunkte hat, um zuvor erlernte Fähigkeiten weiterhin freischalten zu können, muss das Fragment bei einem Wechsel der Spezies zurückgesetzt werden. Ein Zurücksetzen des Fragments wird auch durch die Neuwahl des Berufs oder der Spezialisierung ausgelöst. Ein anderer Beruf sorgt dafür, dass dem Spieler andere Spezialisierungen zur Auswahl stehen. Durch eine neue Spezialisierung können mit den Erfahrungspunkten andere Fähigkeiten als zuvor freigeschaltet werden. Sollte der Nutzer das *ExperienceFragment* aufrufen, ohne eine Spezies, einen Beruf oder eine Spezialisierung festgelegt zu haben, kann er das Fragment nicht nutzen, weil die fehlenden Informationen notwendig für das Verteilen der Erfahrungspunkte und Erlernen von Fähigkeiten benötigt werden. Wenn dieser Fall eintritt, wird ein Dialog geöffnet, der dem Nutzer alle Fragmente auflistet, welche er noch anschließen muss, bevor er das *ExperienceFragment* bearbeiten kann.

AttributeFragment.kt

Das *AttributeFragment* ist lediglich da um dem Spieler seine Attribute anzuzeigen. Dies ist nötig da sie teilweise aus den bereits eingetragenen Daten berechnet werden und der Nutzer sonst keine Möglichkeit diese abzufragen. Dieses Fragment ist aus einer anfänglichen Fehleinschätzung entstanden, da entgegen unserer annahmen der Spieler keinen direkten Einfluss auf die Attribute hat. Dieses Fragment besitzt ebenfalls eine Layoutdatei *char_creation_attribute.xml* um das aussehen zu beschreiben.

MotivationFragment.kt

In diesem Schritt der Charaktererstellung wählt der Spieler die Motivation des Charakters aus. Es gibt insgesamt 30 Motivationen, die in drei Oberkategorien mit jeweils zehn Motivationen eingeteilt sind. Wie auch im *ObligationFragment* hat der Spieler zwei Möglichkeiten zur Auswahl:

1. Der Spieler sucht sich aus den drei Oberkategorien mit insgesamt 30 Einträgen eine für ihn passende Motivation aus.

2. Die App bestimmt durch Drücken eines Buttons per Zufall ein oder zwei Motivationen.

Zur Umsetzung der Listen mit den drei Oberkategorien nutzen wir eine *ExpandableListView*, da sie genau die Eigenschaften mitbringt, die für so eine Darstellung nötig sind. Wenn der Nutzer zu diesem Fragment wechselt, sieht er zunächst nur die drei Oberkategorien Zielsetzung, Beweggrund und Bindung. Diese Group Items besitzen das Layout der *char_creation_career_group_item.xml*, welches ihnen durch den *MotivationListAdapter* zugeteilt wird. Als Unterklasse des *BaseExpandableListAdapters* wird der *MotivationListAdapater* in der *onCreateView*-Methode des Fragments als Adapter der *ExpandableListView* eingesetzt. Klickt der Nutzer auf ein Group Item, wird die Liste mit den zugehörigen Motivationen entfaltet. Das Layout der Child Items ist wiedermal die *char_creation_listview.xml*.

Da die Motivationen in drei Kategorien eingeteilt sind, verlief die Bestimmung der Motivationen per Zufall etwas anders als im *ObligationFragment*. Zunächst wurden anhand einer Tabelle im Regelwerk [Her13, 94] die Wahrscheinlichkeiten der Oberkategorien entnommen. Die Tabelle gibt für jede der Kategorien eine Wahrscheinlichkeit von 30% an. Die übrigen 10% geben die Chance wieder, bei der Erstellung eine zweite Motivation zu bekommen. Tritt dieser Fall ein, müssen zwei Kategorien ausgewürfelt werden. Nachdem eine oder zwei Kategorien bestimmt wurden, aus denen eine Motivation gewählt wird, erfolgt ein erneuter Zufallswurf. Für diese Zufallsbestimmung liegt die Wahrscheinlichkeit für alle Motivationen der jeweiligen Gruppe bei 10%. Auch in diesem Fragment gilt die Zufallsbestimmung als einzige Möglichkeit, eine zweite Motivation zu erhalten. Beim Klicken auf ein Child Item oder das erneute Drücken des Zufallsbuttons wird das Feld für die zweite Motivation zurückgesetzt.

Die *ExpandableListView* war nicht unsere erste Wahl zur Darstellung der Motivationen. Das ursprüngliche Layout des Fragments bestand aus drei untereinander angeordneten *ListViews*, welche durch *TextViews* mit den Namen der jeweiligen Gruppen voneinander abgegrenzt werden sollten. Damit der Nutzer alle drei *ListViews* durch Scrollen erreichen kann, sollten sie in einer *ScrollView* angeordnet sein. Android lässt es allerdings nicht zu, zwei scrollbare Elemente ineinander zu verschachteln. Also musste die einfache Lösung, die bereits im *ObligationFragment* Verwendung gefunden hat, einer komplexeren Lösung weichen. Unser nächster Ansatz war, die drei *ListViews* in *CardViews* zu verpacken und diese dann untereinander anzuorden. Diese Lösung war auch nicht optimal, da die *ListViews* dann auf eine minimal Größe zusammengeschrumpft sind, sodass der Nutzer keine gute Ansicht der Motivationen hatte. Schließlich wurde die Idee der drei *ListViews* komplett verworfen. Der frühere *BaseAdapter* musste durch einen mit wesentlich höherem Implementierungsaufwand verbundenem *BaseExpandableListAdapter* ersetzt werden und die drei *ListViews* wurden in einer *ExpandableListView* zusammengefasst.

AppearanceFragment.kt

Für das Layout dieses Fragments wurde mehr Zeit in Ansrpuch genommen als ursprünglich eingeplant. Zuerst sollten sich alle Elemente in einem *RelativeLayout* befinden und durch mögliche Abhängigkeiten wie *android:layout_below* an ihre angedachte Position geschoben werden. Mit diesem Aufbau des Layouts wurden die *TextViews* jedoch nicht auf einer Linie mit den *EditText*-Elementen gezeichnet, so dass immer ein Höhenunterschied sichtbar war. Der zweite Ansatz bestand aus zwei *LinearLayouts*, die mit Hilfe eines *RelativeLayouts* parallel nebeneinander angeordnet werden und dafür sorgen, dass jeweils ein *EditText*-Element auf der selben Höhe wie das zugehörige *TextView* platziert wird. Das Ergebnis war das selbe wie zuvor. Der dritte Entwurf brachte endlich das erhoffte Ergebnis. Als Root-Layout wird wieder ein *RelativeLayout* genutzt, um die schon erwähnten Abhängigkeiten nutzen zu können. Danach verwenden wir für jedes Paar aus *TextView* und *EditText* ein eigenes *LinearLayout* mit einer horizontalen Ausrichtung. Nur das letzte Paar ist vertikal angeordnet, da die Eingabe im *EditText* eventuell länger werden könnte.

Zusammen mit dem *GearFragment*, das im nächsten Absatz beschrieben wird, ist das *AppearcanceFragment* das einzige Fragment, in dem der Spieler eigene Texteingaben vornehmen kann. Je nachdem, welche Eingabe der Nutzer im jeweiligen *EditText*-Element eintragen soll, wird ihm eine passende Tastatur angezeigt. Verlangt das Eingabefeld z.B. ein Alter für den Charakter, so wird dem Nutzer nur eine Tastatur mit Zahlen als Eingabeoption eingeblendet. Durch diese Eingrenzung der Eingabemöglichkeiten erleichtern wir uns die Datenweitergabe an das Player-Objekt der Activity.

Viel Programmlogik hat dieses Fragment nicht zu bieten, denn es dient lediglich als Eingabemöglichkeit der einzelnen Aussehensoptionen eines Charakters. Sobald eine Eingabe bestätigt wird, aktualisiert das Fragment das zugehörige Attribut des Player-Objekts der Activity.

GearFragment.kt

Ähnlich zum *AppearanceFragment* ist dieses Fragment mit wenig Programmlogik ausgestattet und stellt dem Nutzer *EditText*-Elemente zum Eintragen seiner Ausrüstung zur Verfügung. Der Nutzer kann bis zu drei Waffen und die zugehörigen Schadenswerte eintragen. Zusätzlich gibt es fünf freie Textfelder, in welchen der Spieler seine Ausrüstungsgegenstände und eine kurze Anmerkung zu jedem Gegenstand notieren kann. Da jeder Charakter auch eine gewisse Menge an Credits bei sich trägt, gibt es dazu auch ein eigenes Textfeld.

5.2 Würfeltool

Das Würfeltool stellt einen abgegrenzten Teil unserer Applikation dar. Es wird durch das Hauptmenü(*MAIN_NAVIGATION_DRAWER*) aufgerufen. Wir haben uns auch hier für ein Fragment entschieden, da wir dieses gut in das bestehende System aus *AppBarLayout* und *NavigationView* einbinden konnten. Wir legten zunächst die Klasse *DiceController* an. Diese erbt von Fragment und bildet damit die Hauptkomponente des Würfeltools. Da neben der Hauptansicht zusätzlich noch 2 Dialoge benötigt werden erstellten wir 3 Layout-Dateien(*fragment_dice.xml*, *dice_result_display_dialog.xml*, *dice_roll_history.xml*). Um das Fragment strukturiert zu gestalten wurde der Inhalt durch CardViews in die drei Teile "Verfügbare Würfel", "Ausgewählte Würfel" und "Buttons" aufgeteilt. Zum Beginn der Implementierung zeigte sich, dass aufgrund des Umfangs, eine weitere Klasse *DiceHelper* angelegt werden muss. In dieser implementierten wir alle Methoden und Daten die für die Würfel benötigten wurden einschließlich aber nicht ausschließlich Wahrscheinlichkeiten, Würfelarten und Zufallsgenerierung. Hierbei stellt die innere Klasse *DiceSide* eine Würfelseite da, sie hält für jedes Symbol einen Integer-Wert der die Anzahl beschreibt. Für jeden Würfel wurde dann ein Array angelegt welches die entsprechende Anzahl an Seiten in Form der oben genannten Klasse speichert. Insgesamt gibt es sieben verschiedene Würfel und 8 unterschiedliche Symbole. Zusätzlich gibt es eine Klasse *Dice* welche diese Arrays verwendet um so mit einer Zufallsmethode eine Würfelseite zurückgeben zu können.
Auch im DiceController wurden zusätzliche innere Klassen benötigt um den Würfelverlauf korrekt zu speichern und darzustellen. Die Einträge des Verlaufs werden durch *RollHistoryEntry* mittels der Zeit und *DiceSide* repräsentiert. Jedes Würfelsymbol erhält zum Aufruf des Fragmentes einen *onClickListener* welcher sobald der Nutzer auf die jeweilige Grafik tippt, einen Würfel der entsprechenden Kategorie zum "Pool" hinzufügt. Die maximale Anzahl pro Typ wird dabei in der Variable *diceMaxCount* festgehalten und beträgt 6, da das Regelwerk nicht mehr erlaubt. Wenn ein Würfel zum Pool hinzugefügt wird so wird in dem Array *diceCount* der entsprechende Wert um 1 erhöht. Im Array befinde sich für jeden Würfeltyp ein Integer welches die Anzahl der momentan zum Pool hinzugefügten speichert. So kann dynamisch ein Element entfernt und hinzugefügt werden. Sollte der Benutzer sich entscheiden zu Würfel wird entsprechend oft die random-Method der Dice Klasse aufgerufen. Da diese ein DiceSide-Object zurückliefert, kann man alle Ergebnisse addieren und so das Gesamtergebnis in einem DiceSide-Object speichern. Abschließend werden noch Symbole, welche sich gegenseitig aufheben, gegeneinander gerechnet und alles mit dem aktuellen Zeitstempel in die *RollHistory*-ArrayList gespeichert. In dieser werden immer die letzten 5 Ergebnisse gespeichert. Um den Würfelverlauf dauerhaft speichern zu können verwendeten wir *SharedPreferences* und die JSON-Bibliothek Google Gson [Goo13]. Zunächst wurde das ArrayList-Object in eine JSON-repräsentation umgewandelt und diese dann als String in die Preferences Datei eingetragen. Zum laden des Objektes wird dieser Prozess umgekehrt. Abschließend sei zu sagen das

die größte Herausforderung bei der Implementierung das entwickeln einer passenden Datenrepräsentation der Würfel war. Die Wahrscheinlichkeiten uns Symboltabellen entnahmen wir dem *Under a black Sun*-Storybook [Ost13].

5.3 Import/Export

Die Möglichkeit den Charakter zu importieren und auch zu exportieren war uns sehr wichtig, da die Charaktere auch auf anderen Geräten zur Verfügung stehen sollten. Da wir für das speichern in dem Projektrahmen keine eigene Infrastruktur zur Verfügung stellen konnten, haben wir uns über alternativen informiert. Schließlich ist die Wahl dann auf Google Firebase gefallen, da hier eine bereits bestehende Java Bibliothek angeboten wurde und der Dienst in vielen Android-Applikationen Anwendung findet. Um den Firebase nutzen zu können muss zunächst auf der entsprechenden Website ein neues Projekt angelegt werden. Um ein solches zu erstellen braucht man als Entwickler lediglich den Paketnamen und kann sich dann eine Konfigurationsdatei herunterladen. Anschließend werden die Abhängigkeiten in die *build.gradle* Dateien eingetragen. Nun ist die Einrichtung abgeschlossen und im Quellcode kann auf die *Firebase API* zugegriffen werden. Der Nutzer kann sich, sofern noch nicht geschehen, einloggen und dann seinen Charakter herunter- bzw. hochladen. Außerdem bekommt er eine Liste mit den gespeicherten Charakteren angezeigt. Das aussehen dieses Fenster wurde in der *fragment_import_export.xml* festgelegt. Diese wird über den Standard *LayoutInflater* eingelesen und dann als Fragment dargestellt. Abschließend sei zu dieser Funktion zu sagen, dass die Implementierung deutlich einfacherer war als ursprünglich angenommen. Durch die gute Unterstützung von “Google Firebase“ konnten wir das geplante dann schnell umsetzen.

6 Test und Usability

Der Code wurde während der Entwicklung auf den in der Tabelle aufgelisteten Geräten getestet.

Da nicht genügend unterschiedliche Geräte als Hardware vorhanden waren, wurden die Nexus Smartphones emuliert. Hierbei haben wir auf eine große Varianz bei den

Gerät	Android Version
Motorola Moto G4	7.1
Samsumg Tab A	6.0
Nexus 5X	8.1
Nexus 5X	5.0
Nexus 6P	8.1

Tabelle 1: Testgeräte

Displayproportionen geachtet und verschiedene SDK Versionen verwendet. Nach jeder größeren Änderung im Programmcode wurde der Code im Emulator von Adroid Studio getestet. Wenn ein Feature fertiggestellt wurde, haben wir die Funktionen auf unseren Geräten ausgiebig getestet. Aufgrund der geringen Teamgröße von zwei Personen hatte jeder dauerhaft einen Überblick des aktuellen Implementierungstandes. Da das Konzept die Kapselung der Module bis zu einem gewissen Grad nicht erlaubt, konnte auf modulbeshränkte BlackBox Tests verzichtet werden.
Um zusätzlich fachliche Mängel ausschließen zu können, haben wir verschiedene Iterationen der Applikation mit einer Anwendergruppe bestehend aus Personen der Zielgruppe getestet. Die 6 Personen haben unser Produkt in einem realistischen Szenario(Pen und Paper Abend) verwendet und anschließend ihre Erfahrungen in Einzelgesprächen mitgeteilt.

7 Zusammenfassung

In diesem Kapitel fassen wir zunächst einzeln unsere Erfahrungen zusammen, abschließend wird dann das Entwicklungspotenzial der Applikation bzw. Idee näher betrachtet.

7.1 Fazit

Christopher Kluck

Zu Beginn des Projekts war ich eher skeptisch der neuen Programmiersprache und der ungewohnten Plattform gegenüber. Inzwischen habe ich mich an Kotlin gewöhnt und würde es aufgrund der Vorteile Java gegenüber in der App-Entwicklung unter Android vorziehen. Durch Android-Studio wird einem der Einstieg in die Android-Programmierung deutlich erleichtert. Der eingebaute Emulator ließ sich auf meinem Laptop zwar nicht starten, aber am Desktop-PC hat er das Testen für mich überhaupt erst möglich gemacht, da ich kein Android Gerät besitze. Das Projekt hat mir gezeigt, dass das Programmieren für mobile Endgeräte zu einem großen Anteil aus GUI-Design besteht, schließlich soll die App auf möglichst vielen Smartphones gut aussehen und benutztbar sein. Normalerweise meide ich alles, was mit GUI-Design zutun hat, weswegen mir dieser Teil des Projekts eher weniger Spaß gemacht hat. Nachdem die GUI fertig war, ging es jedoch ans Schreiben der Funktionen, die im Hintergrund ablaufen. Dieser Teil der Entwicklung war für mich deutlich angenehmer. Während der Entwicklung habe ich ständig neue Dinge gelernt, auf die man während der Implementierung achten sollte, damit der Nutzer am Ende eine gut funktionierende App geliefert bekommt. Auch, wenn es während des Projekts Phasen gab, in denen ich wenig motiviert war an der App zuarbeiten, bin ich mit dem Verlauf der Entwicklung zufrieden. Abschließend kann ich sagen, dass ich viel über die App-Entwicklng gelernt habe, sei es durch die Vorträge während

der Vorlesung, durch gezieltes Selbstudium zu bestimmten Bereichen oder durch zufällige Entdeckungen während der Fehlerbehebung mancher Probleme.

Philipp Clausing

Durch die Entwicklung der Applikation konnte ich sehr viel über Android erfahren. Ich habe mitbekommen wie aufwendig die Planungsphase, auch bei vermeintlich trivialen Funktionalitäten, sein kann. Eine Besonderheit bei der Erstellung von Mobilen Applikationen ist der Fokus auf die Benutzeroberfläche, so musste viel Zeit bei der Erstellung dieser aufgewendet werden. Eine weitere Hürde für mich war die "Beständigkeit" von Daten bzw. der Lebenszyklus von Activity's , Fragmenten, da dies im Kontrast steht mit der traditionellen Desktopentwicklung. Unsere Teamarbeit kann ich nur positiv bewerten, wir haben zu keinem Zeitpunkt große Unstimmigkeiten in unserer Vision der Applikation gehabt und konnten dem jeweils anderen immer mit Rat und konstruktiver Kritik zur Seite stehen. Schließlich ziehe ich eine positive Bilanz und kann mit neuem Wissen zukünftige Aufgaben angehen.

7.2 Entwicklungspotenzial

Unter Entwicklungspotenzial möchten wir die Punkte näher beleuchten welche im Rahmen des Projektes vernachlässigt wurden, da Sie den angesetzten zeitlichen Rahmen gesprengt hätten.

- **Gruppenfunktionen**
  Da man bei einem Pen und Paper Rollenspiel immer in einer Gruppe agiert, wäre eine Funktion zum synchronisieren von Spielen oder das zusammenfassen zu einer Gruppe ein passendes Feature. Auch das gemeinsame Planen von Terminen ist hier denkbar.

- **Weitere Regelwerke**
  Die Möglichkeit andere Regelwerke zu unterstützen haben wir seit beginn angedacht. Allerdings aufgrund der unterschiedlichen Komplexität und dem Mehraufwand bei der Architekturplanung nicht umgesetzt. Sollte man die Applikation unabhängig weiter entwickeln, wäre dies ein geplantes Feature.

- **Zusätzliche Plattformen**
  Die von uns entwickelte App würde auch auf anderen Gerätetypen Anwendung finden. Denkbar wären hier iOS, Windows und Linux sowie ChromeOS.

Literatur

[Aue] Martin Auer. Free UML Tool for Fast UML Diagrams. `http://www.umlet.com/`
Stand 22.01.2018.

[Goo13] Google. A Java serialization/deserialization library to convert Java Objects into JSON and back. 2013. `https://github.com/google/gson`
Stand: 18.01.2018.

[Her13] Sterling Hershey. *Star Wars: Edge of the Empire Core Rulebook*. Fantasy Flight Games, 2013.

[Kaz17] Piotr Kazmierczak. Why Kotlin Is the Hero Android Needs. *10clouds*, 2017. `https://10clouds.com/blog/kotlin-android/`
Stand 17.01.2018.

[kot16] Statically typed programming language for modern multiplatform applications. 2016. `https://kotlinlang.org/`
Stand 17.01.2018.

[Mun] Florina Muntenescu. Android Architecture Patterns Part 1: Model-View-Controller. `https://upday.github.io/blog/model-view-controller/`
Stand 22.01.2018.

[Ost13] Katrina Ostrander. *Under a black sun*. Fantasy Flight Games, 2013.

[Par17] Eugen Paraschiv. Data Classes in Kotlin. *Baeldung*, 2017. `http://www.baeldung.com/kotlin-data-classes`
Stand: 17.01.2018.