ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Institut für Datenbanken und Informationssysteme

Mobile Application Lab - FancyQuartett Dokumentation

Dokumentationan der Universität Ulm

Vorgelegt von:
Lukas Stöferle, Ferdinand Birk, Marius Kircher
lukas.stoeferle@uni-ulm.de, ferdinand.birk@uni-ulm.de, marius.kircher@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert
Marc Schickler

Betreuer:
Marc Schickler

WS 2015/16

Fassung 23. März 2016

Inhaltsverzeichnis

1

Einleitung

Seit der Ankündigung des weltweit ersten Smartphones durch *Apple Inc.* im Jahre 2007 hat sich sehr viel auf dem Markt für mobile Endgeräte getan. Heute gibt es eine Vielzahl an verschiedenen Smartphones von unterschiedlichen Herstellern, die jeweils mit unterschiedlichen Betriebssystemen ausgestattet sind, wie z.B. *Android* von *Google Inc.*, *iOS* von *Apple Inc.* oder *WindowsPhone* von *Microsoft*. Smartphones sind heute nicht mehr aus unserem Alltag wegzudenken, da sie den Benutzer aufgrund ihrer Mobilität und dank verschiedener Applikationen (kurz „Apps“) unterstützen aber auch unterhalten. Im diesjährigen Anwendungsfachs *Mobile Application Lab* entstand deshalb im Rahmen eines kleinen Softwareprojektes eine Quartett-App, die sich durch eine beliebige Anzahl an auswählbaren Kartendecks von den bereits existierenden Apps im App-Store abheben soll. Hierbei werden Kartendecks per *Representational State Transfer (REST)* von einem Server der Universität Ulm heruntergeladen, die dann auf dem Gerät persistent gespeichert werden und somit auch offline verfügbar sind. Neben den vorgegebenen Anforderungen durfte jedes Team eigene Ideen einbringen, was zu unterschiedlichen Anwendungen führte.

1.1 Motivation

Apps spielen heute, aber auch in Zukunft eine große Rolle im mobilen Software-Bereich. Durch die Programmierung einer Quartett-App in Teams soll mehr Praxiserfahrung in das doch eher theorielastige Studium mit einfließen. Die Quartett-App deckt möglichst viele Bereiche der App-Programmierung ab, wodurch die Teams später in der Lage sind, ihre eigenen Ideen und Anwendungen planen und umsetzen zu können.

1.2 Kontext

Die Quartett-App wird im Auftrag des *Institut für Datenbanken und Informationssysteme* der *Universität Ulm* entwickelt. Dabei sollen die Teams grundlegende Werkzeuge zur App-Programmierung kennenlernen und somit Schritt für Schritt die benötigten Schritte der App-Entwicklung meistern. Durch sorgfältige Planung, wie z.B. mit Hilfe von Mockups oder Diagrammen, soll die spätere Implementierung vereinfacht werden. Neben der Planung und Implementierung der Anwendung sollen die Teams sich mit den Besonderheiten der Zielplattform und der dazugehörigen Smartphones (wie z.B. Datenhaltung, Sensorik oder gestenbasierter Eingabe) auseinander setzen.

2

Anforderungen

In diesem Teil der Dokumentation werden die Anforderungen, welchen die App entsprechen soll, aufgeführt und erläutert. Dabei werden sowohl die Anforderungen unseres Betreuers als auch die eigenen optionalen Anforderungen zusammengefasst.

2.1 Funktionale Anforderungen

2.1.1 Spieltyp

Die Anwendung soll ein reines Single-Player-Spiel sein, d.h. es gibt keine Möglichkeit per Bluetooth- bzw. Internetverbindung mit anderen Geräten zu kommunizieren. Der Benutzer spielt also immer gegen den Computergegner. Es soll lediglich die Möglichkeit geben zu zweit am selben Gerät ein Multiplayer-Spiel (Hotseat) zu spielen, somit können zwei Personen gegeneinander spielen.

2.1.2 Spielmodi

Dem Spieler soll eine Auswahl von drei verschiedenen Spielmodi angeboten werden, welche im Folgenden erläutert werden:

To-The-End Der Spieler, der zum Schluss alle Karten besitzt, hat das Spiel gewonnen. Ein Spieler hat das Spiel verloren, wenn er keine Karten mehr besitzt.

Time Ein Spiel ist durch ein Zeitlimit begrenzt. Wird dieses Limit erreicht, gewinnt der Spieler der die meisten Karten besitzt. Falls beide Spieler gleich viele Karten besitzen sollten wird eine extra Runde gespielt, in welcher dann der Sieger bzw. Verlieren gekürt wird. Hat ein Spieler vor Erreichen des Zeitlimits keine Karten mehr, hat er verloren.

Points Jeder Spieler besitzt ein Punktekonto, welches sich durch gewonnene Duelle erhöht. Erreicht ein Spieler die maximale Punktzahl hat er das Spiel gewonnen. Besitzt ein Spieler keine Karten mehr, bevor die maximale Punktzahl erreicht wird, gewinnt der Spieler der die meisten Punkte besitzt. Die Punkte ergeben sich aus der Qualität der gewählten Attributwerte der aktuellen Karte. Sehr gute Attribute geben nur 1 Punkt, Attributewerte die eine $\pm$5% Differenz zum Median des Wertes im Kartendeck besitzen geben 2 Punkte, die schlechteren Attributwerte ergeben 5 Punkte. Gewinnt der Gegenspieler ein Duell, so bekommt er die Punkte die er für seinen Attriutwert bekommen hat gutgeschrieben. Dieser Spielmodus soll vor allem den Spieler dazu verleiten, schlechte Attributwerte zu wählen.

2.1.3 Einstellungsmöglichkeiten

Zusätzlich zu den drei Spielmodi sollen weitere Einstellungen möglich sein, welche das Spielerlebnis abwechslungsreicher machen sollen. Diese sind optional und können je nach belieben aktiviert bzw. deaktiviert werden.

Maximale Rundenanzahl Ein Spiel kann durch eine maximale Rundenanzahl begrenzt werden. Wird das Rundenlimit erreicht, wird ein Gewinner bzw. Verlierer ermittelt. Wenn es keinen Gewinner gibt wird eine zusätzliche Runde gespielt, in welcher dieser dann ermittelt wird.

Zeitlimit für ein Spielzug Der Spielzug eines Spielers kann durch ein Zeitlimit begrenzt werden. Wählt ein Spieler nach Ablauf der Zeit keinen Attributwert aus, so wird zufällig ein Attributwert der aktuellen Karte ausgewählt.

2.1.4 Gespeichertes Spiel

Während eines Spiels soll immer die Möglichkeit bestehen, das Spiel zu pausieren. Hierbei soll das aktuelle Spiel gespeichert werden, welches dann im Hauptmenü später fortgesetzt werden kann. Das pausierte Spiel soll persistent gespeichert werden, damit es nach einer Schließung der App weiterhin existert.

2.1.5 Statistiken

Es soll die Möglichkeit bestehen, eine kleine Statistik über die vergangenen Spiele einsehen zu können. In dieser sollen die Anzahl der gespielten Spiele und die Anzahl der bisherigen Duelle aufgelistet werden, welche jeweils die Anzahl der Siege bzw. Niederlagen beinhalten.

2.1.6 Galerie

In der Anwendung soll es eine Galerie geben, in welcher die offline und online verfügbaren Kartendecks aufgelistet werden. Zusätzlich soll hier die Möglichkeit bestehen, dass weitere Kartendecks heruntergeladen werden können. Die Ansicht der Kartendecks und der Karten kann zwischen einer Listen- und einer Grid-Ansicht gewechselt werden. Außerdem soll es eine Detailansicht für einzelne Karten geben in welcher diese dann genauer betrachtet werden können. In der Detailansicht soll dann mit Wischgesten nach links bzw. recht die vorherige bzw. nächste Karte angezeigt werden.

2.1.7 Kartendecks

Die App soll in der Lage sein mehrere Kartendecks zu verwalten, mit welchen der Spieler dann spielen kann. Ein Kartendeck besteht aus mindestens 8 und höchstens 72 Karten. Die Karten setzen sich aus einem Titel, mindestens einem Bild und beliebig vielen Attributen zusammen. Ein Attribut besteht aus einem Namen, Wert und der zugehörigen Einheit. Zusätzlich können Attribute optional ein Icon beinhalten. Um unterscheiden zu

können ob der höhere oder niedrigere Attributwert gewinnt wird anhand einer „WhatWins-Variable“ bestimmt.

2.1.8 Serverkommunikation

Der Benutzer kann in der Galerie weitere Kartendecks online beziehen. Hierbei sollen in der Galerie vorerst minimale Vorschaudecks vom REST-Server heruntergeladen und angezeigt werden. Damit kann Datenvolumen gespart werden, was beim mobilen Netzwerk von großer Bedeutung sein kann. Der Benutzer kann dann ein Kartendeck wählen, welches vom Server heruntergeladen und auf dem Gerät offline gespeichert wird.

2.2 Nicht-funktionale Anforderungen

2.2.1 Material Design

Die Anwendung soll sich nach an die Vorgaben zum *Material Design* von *Google Inc.* halten. Die grafische Elemente sollen dadurch einen hohen Wiedererkennungswert haben was wiederum die Usability der Anwendung zu verbessert.

2.2.2 Gestensteuerung

Die Anwendung soll verschiedene gestenbasierte Eingaben unterstützen. Speziell sollen die Gesten „Wischen“ und „Schütteln“ verwendet werden. Gestensteuerung ist eine Besonderheit von mobilen Geräten, daher bietet es sich an derartige Funktionen mit einzubinden.

2.2.3 Animationen

Animationen sollen dazu verwendet werden, um das „Look & Feel“ der Anwendung zu verbessern. Außerdem sollen sie gezielt die Aufmerksamkeit des Benutzers auf bestimmte Ereignisse lenken.

2.2.4 Offline-Szenario

Ist keine Verbindung zum Internet möglich, sollen trotzdem alle Funktionen die nicht zwingend eine Internetverbindung benötigen zur Verfügung stehen.

3

Architektur

Die Software-Architektur unserer App musste sich in erster Linie den Besonderheiten des Android-Systems anpassen. Dazu wurden neben den Java Paketen hauptsächlich die spezifischen Android Bibliotheken benutzt.

Um die funktionalen Anforderungen zu erfüllen, wurde ein Datenmodell zur Speicherung während der Laufzeit sowie zur persistenten Speicherung benötigt. Die wichtigsten Datenobjekte bei dem Quartett-Spiel sind *Deck* sowie *Card*, wobei ein Deck aus mehreren Karten und eine Karte aus mehreren Bildern sowie Attributwerten besteht.

Eine zentrale funktionale Anforderung ist die Anbindung an einen REST-Server, von dem neue Kartendecks geladen werden können. Die Architektur benötigt also ein Modul, das HTTP-Anfragen senden, textbasierte Daten empfangen und diese in das interne Datenformat umwandeln kann.

Die eben genannten Aspekte werden im Folgenden näher erläutert.

3.1 Besonderheiten in Android

Eine Android-App setzt sich aus einer oder mehreren *Activities* zusammen. Außerdem ist der App eine *AndroidManifest.xml* Datei zugeordnet, die die enthaltenen Activities deklariert und in eine hierarchische Beziehung zueinander stellt. Activities werden als Java-Klassen implementiert, die von der Klasse *Activity* aus der Android Bibliothek ableitet. Den Einstiegspunkt in die App bildet die *MainActivity*.

3.1.1 Activity Lifecycle

Das Speichermanagement wird vom Android-System im Hintergrund geleistet. Aus Speicherplatz- und Energieeffizienzgründen auf mobilen Geräten haben Activities einen sogenannten Lifecycle. Zu entsprechenden Zeitpunkten werden vom System die Methoden *onCreate()*, *onPause()*, *onStop()* usw. aufgerufen (siehe Abbildung 3.1). In diese Methoden wird vom Anwendungsprogrammierer Code eingefügt, hierdurch kann er bei auftreten der entsprechenden Zustandsübergänge gewünschtes Verhalten implementieren und so z.B. aktuelle Daten vor dem Schließen der Anwendung persistieren.

3.1.2 Layouts und Views

Um eine GUI anzuzeigen benötigt die Activity ein Layout. Das Layout wird i.d.R. in *onCreate()* mit dem API *setContentView()* gesetzt. Layouts werden in XML Resourcen spezifiziert. Ein Layout enthält hierarchisch angeordnete *Views* (dazu gehören Buttons, TextViews, Checkboxes, ProgressBars, ImageViews usw.). Views erhalten Attribute, die ihre Größe und Position im Layout und ihr Erscheinungsbild festlegen. Außerdem können sie ID-Attribut enthalten, mit dem in der Activity auf sie zugegriffen werden kann.

Die Methode *findViewById()* liefert eine Referenz auf die entsprechende View-Instanz. Diese kann nun programmatisch verändert werden. Views implementieren das Observer-Pattern und ermöglichen damit ereignis-orientierte Programmierung. Die Klasse *View* (bzw. davon abgeleitete Klassen) nehmen Listener-Objekte entgegen, welche Methoden enthalten, die bei dem entsprechenden Ereignis aufgerufen werden.

3.1.3 Kommunikation zwischen Activities

Zu einem Zeitpunkt ist i.d.R. nur eine Activity aktiv. Beim Wechsel von einer Activity zu einer anderen müssen jedoch Informationen übergeben werden können. Der Wechsel erfolgt zentral durch die Methode*startActivity()*. Sie erhält ein *Intent*-Objekt, welches die Informationen in serialisierter Form enthält. Zu übergebende Objekte müssen hierzu das *Serializable*- oder *Parcelable*-Interface implementieren.

3.1.4 Activities in FancyQuartett

Insgesamt werden 8 Activities verwendet, deren Struktur Abbildung 3.2 zeigt, welche hier kurz erläutert werden.

Die *MainActivity* besteht aus 3 Fragmenten. Fragmente sind *Sub-Activities*, die in eine Activity eingebettet sind und haben ihren jeweils eigenen Lifecycle. Sie können beispielsweise in einem sogenannten *ViewPager* als Tabs verwendet werden.

Das erste Fragment dient als GUI zum starten eines Spiels. Startet der Benutzer ein neues Spiel, wird zur *NewGameSettingsActivity* gewechselt. Diese dient zum wählen eines Decks und der Spieleinstellungen. Zum wählen eines Decks wird die *NewGameGalleryActivity* gestartet, die alle verfügbaren Decks (sowohl offline als auch online) anzeigt. Nach dem Klick auf Start wird zur *GameActivity* gewechselt. Diese stellt die aktuelle Karte dar und instanziiert die Klasse *GameEngine*, welche die Ausführung der Spiellogik übernimmt.

Das zweite Fragment stellt die Deck-Gallerie dar. Bei der Auswahl eines Decks wird die *CardActivity* gestartet, die alle Karten des Decks in einer *ListView* anzeigt. Wird eine Karte ausgewählt wird die *CardViewerActivity* gestartet, die die Karte mit ihren Bildern und Attributwerten in einem *ViewPager* anzeigt. Durch *Swipen* nach links bzw. nach rechts kann zur vorherigen bzw. nächsten Karte des Decks navigiert werden.

Das dritte Fragment zeigt eine Statistik an und bietet keine Interaktionsmöglichkeiten.

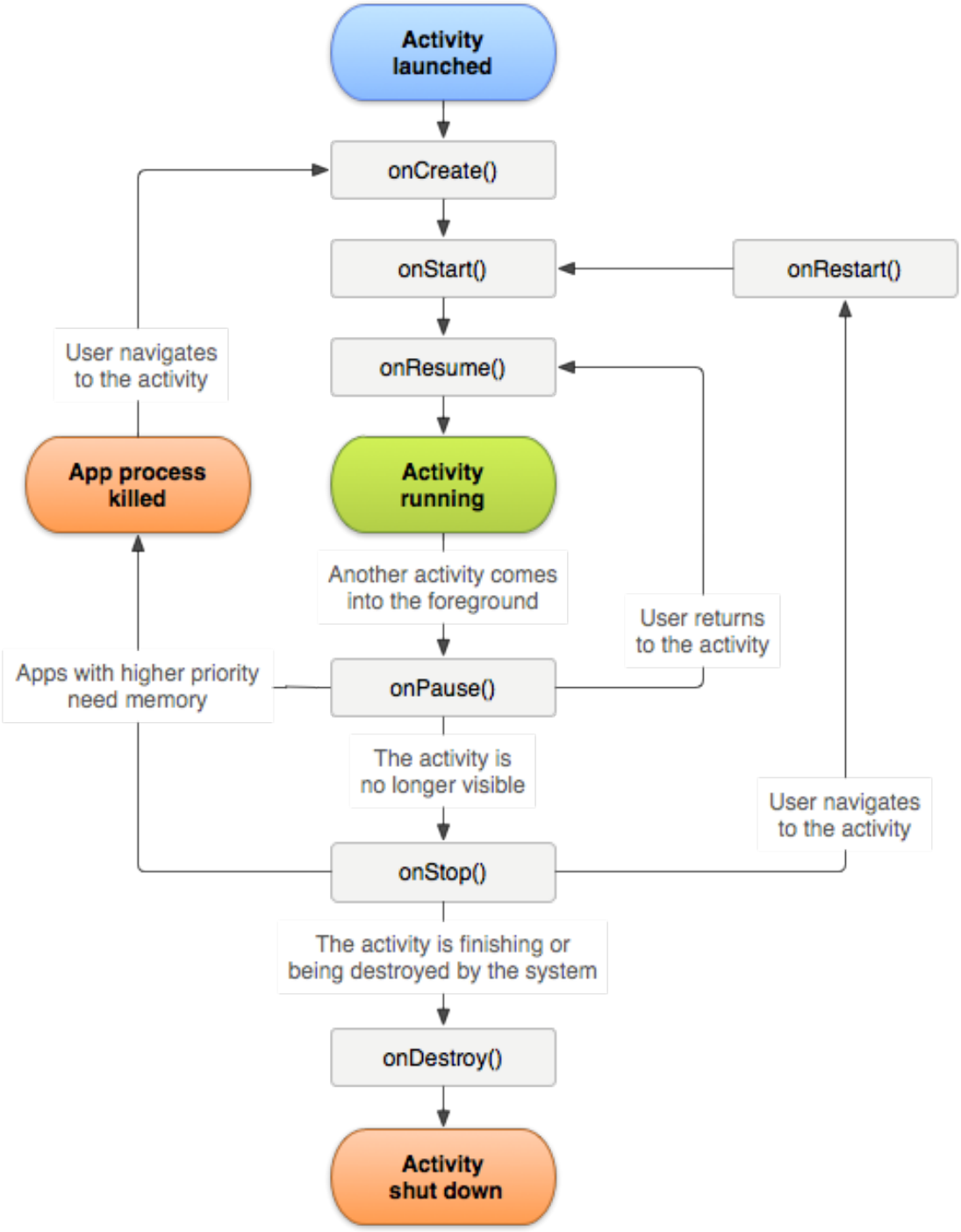

Abbildung 3.1: Activity Lifecycle

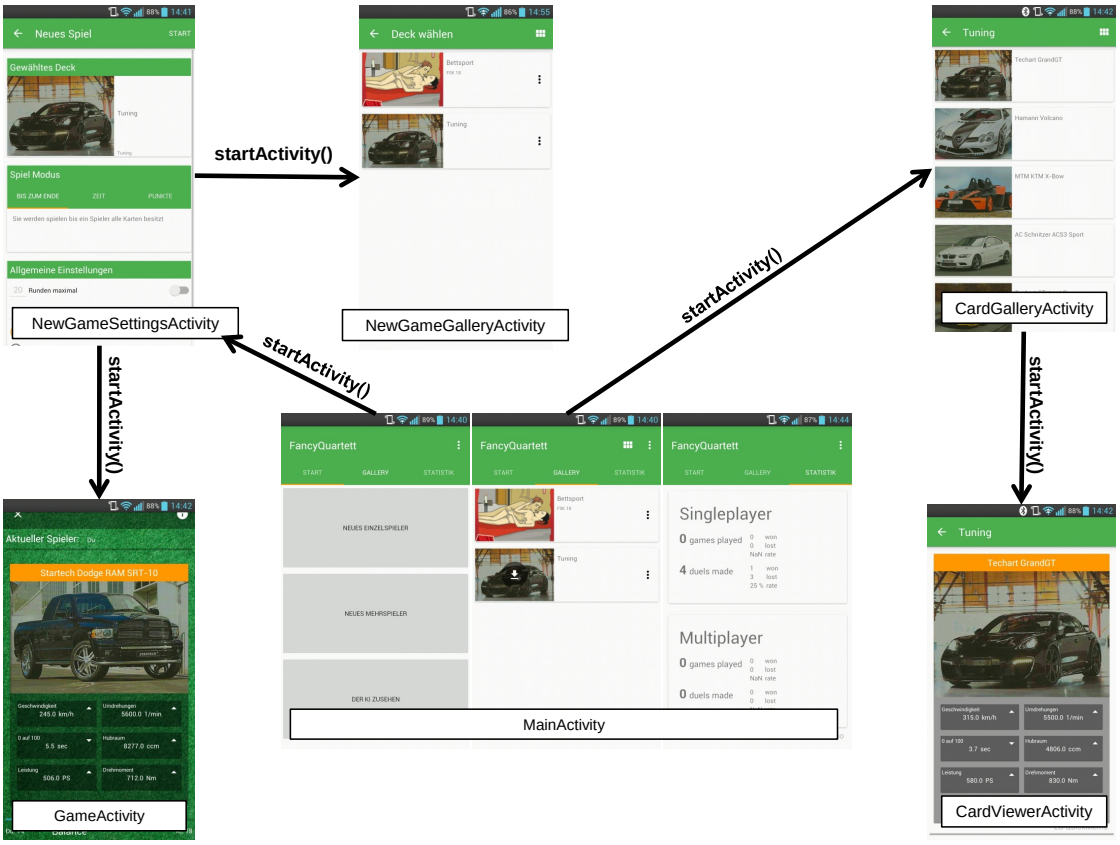

Abbildung 3.2: Activities in FancyQuartett

3.2 Datenmodell

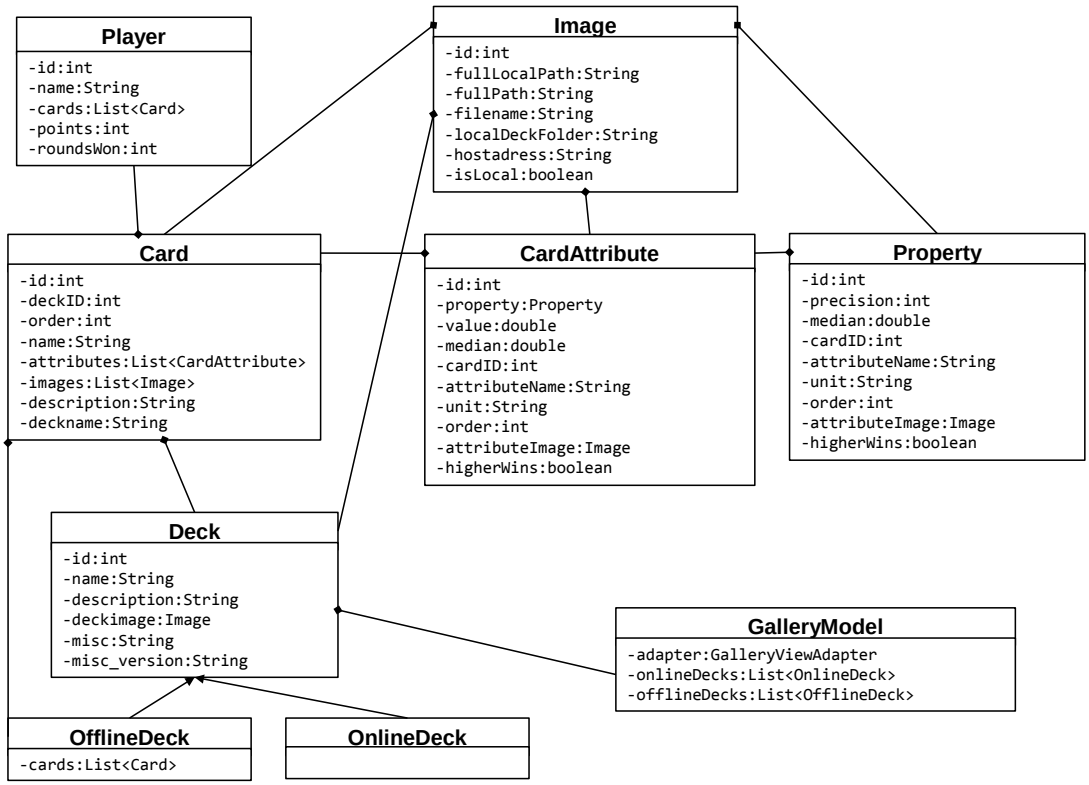

Abbildung 3.3: Datenmodell von FancyQuartett

Abbildung 3.3 zeigt die Realisierung der Datenbasis von FancyQuartett. In dieser Form werden die Daten zur Laufzeit gehalten.

Die persistente Speicherung erfolgt auf dem Dateisystem. Android weist jeder App einen Bereich im Dateisystem zu. Dort wird mittels der Klasse *DeckDownloader* für jedes heruntergeladene Deck einen Ordner angelegt, in dem eine JSON Datei und die zu den Karten gehörenden Bilder gespeichert werden. *DeckDownloader* erbt von der Klasse *AsyncTask*. Das ermöglicht es, die Ein-/Ausgabe-Operationen in einem separaten Thread (asynchron) auszuführen, um die GUI nicht zu blockieren.

3.3 Netzwerkfunktionen

Für das Laden von Decks über das Netzwerk wurde ein Server mit einer REST-API zur Verfügung gestellt, der Daten im JSON-Format sendet. Von diesem konnten folgende Ressourcen abgefragt werden:

- /**decks** liefert ein Array aus Objects. Die Anzahl der Objects entpricht den verfügbaren Decks. Die Objects enthalten jeweils die für ein Deck relevanten Informationen, die den Attributen im obigen Klassendiagramm in Abbildung 3.3 entsprechen.
- /**decks**/**<ID>** liefert das Deck-Object mit der entprechenden ID.
- /**decks**/**<ID>**/**cards** liefert ein Array aus Objects, die jeweils die ID und den Namen einer Karte enthalten.
- /**decks**/**<ID>**/**cards**/**<ID>** liefert das Card-Object mit der entsprechenden ID. Die Attribute in diesem Object nicht enthalten.
- /**decks**/**<ID>**/**cards**/**<ID>**/**attributes** liefert ein Array aus Objects, die jeweils die für ein Attribut und dessen Wert relevanten Informationen enthalten.

Durch den Aufruf von *openConnection()* auf den entsprechenden URLs wird ein *HttpURLConnection*-Objekt zurückgegeben, auf welchem noch der erforderliche Authorization Header gesetzt werden muss um Zugang zur API zu erhalten. Die empfangenen Daten können bei einer erfolgreichen Anfrage aus einem *InputStream* gelesen werden.

Das Package *org.json* macht das Handling der empfangenen JSON-Daten einfach. Es bringt für JSON spezifizierte Klassen und entsprechende Operationen mit. Ein *JSONObject* bzw. *JSONArray* kann direkt aus einem ASCII-String erstellt werden. Die Konvertierung in das interne Datenformat (siehe Abbildung 3.3) erfolgt durch Aufrufe auf den Klassenkonstruktoren, wobei die Werte aus den JSON-Objekten übergeben werden.

4

Implementierung

Die Anwendung wurde in *Android Studio 1.5.1* implementiert. Zur Versionsverwaltung wurde ein bei Github gehostetes Git-Repository verwendet. Als Unterstützung beim Anbinden der REST-API wurde die Google Chrome-Erweiterung *Advanced Rest Client (ARC)* verwendet, welche es erlaubt RAW-HTTP Requests abzusetzen und anschließend die Antwort in Plaintext anzuzeigen.

4.1 Vorgehensweise

In der Implementierungsphase wurden zuerst die Datenklassen implementiert und mit den Standard Getter und Setter ausgestattet. Darauf aufbauend wurde die Implementierung in Gameentwicklung, Deckmanagemant und Statistikmanagemant eingeteilt und fortan parallel implementiert.

4.2 Datenhaltung

Für die Speicherung der Deckdaten, welche aus Text- und Bilddaten bestehen, haben wir uns für die Variante des Dateisystems im internen Speicher entschieden. Da wir die Deckdaten als JSON erhalten, bietet es sich an, diese nach spezifizierten Modifikationen auch wieder als JSON zu speichern und erreichen so eine gewisse Konsistenz in den *OfflineDeck- und OnlineDeckLoadern*. Es wäre ein nicht unwesentlicher Mehraufwand gewesen, eine entsprechende Datenbankklasse zu schreiben, die mittels SQL-Statements die JSON-Daten in einem eigenen Schema persistiert speichert.

Für die Speicherung der Spielzwischenstände und der Statistiken wird Androids Konzept der *Shared Preferences* genutzt. Dies sind Key-Value Paare welche in dem *shared_prefs* Ordner einer jeden App als XML gespeichert werden. Sollen Objekte abgespeichert werden wie z.B. die Spielzwischenstände, dann müssen diese Objekte serialisiert sein. Da wir zum Speichern des aktuellen Spielzustandes ganz einfach das *GameEngine*-Objekt abspeichern impliziert dies, dass unsere *GameEngine*-Klasse das Interface *Serializable* implementiert. Das betrifft nicht die Controller der GameEngine, weshalb diese bei jedem "Resume"neu initialisiert werden müssen, da sie nicht mit serialisiert werden.

Ein heruntergeladenes Deck wird wie in Abbildung 4.1 auf dem Dateisystem abgelegt. Durch diese exakt spezifizierte Struktur ist es möglich einfache *LocalDecksLoader* bzw. *LocalDeckLoader* zu implementieren. Da die Daten intern unter

/data/data/[packagename]/files/

gespeichert werden kann eine manuelle Veränderung der Daten ohne gesonderte Root-Rechte ausgeschlossen werden und daher für den Zweck dieses Spiels als geeignet angesehen werden.

Die Klasse *Image* aus Abbildung 3.3 ist nicht wirklich ein Bild sondern lediglich eine Wrapper-Klasse für die Informationen des zugehörigen Bildes. Hier wird zur Laufzeit festgehalten, ob und wo das Bild lokal ist, sowie die Onlineadresse wo es im Zweifel nachgeladen werden kann. Die Klasse enthält eine Methode *getBitmap()* welche dann aus dem lokalen Pfad mit der Klasse *BitmapFactory* ein *Bitmap*-Objekt decodiert und zurückgibt.

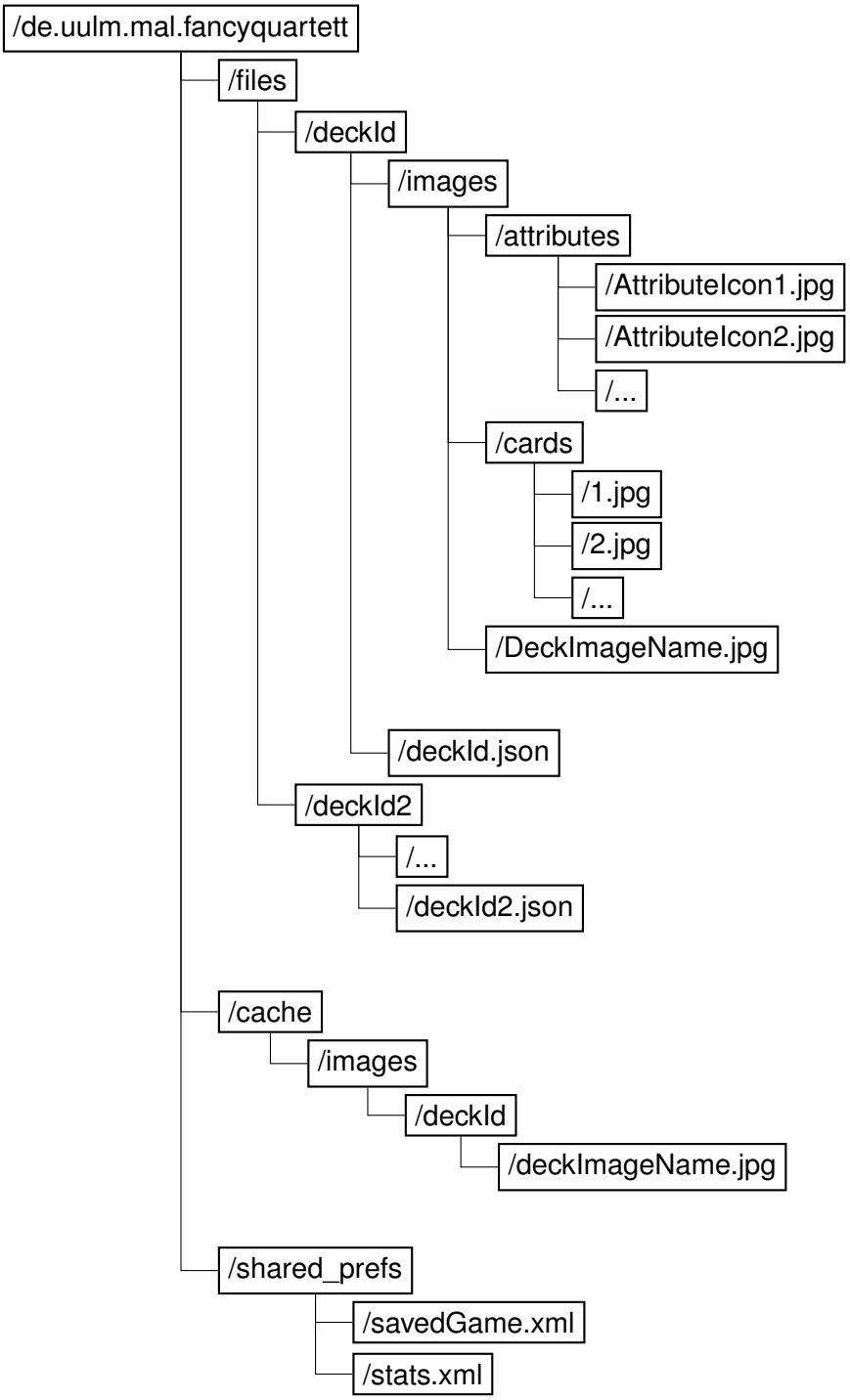

Abbildung 4.1: Ordnerstruktur im Dateisystem

4.3 Datenverarbeitung

In der Implementierung der Datenklassen für die Decks wird wie in Abbildung 3.3 zu sehen zwischen *OfflineDeck* und *OnlineDeck* unterschieden. So ist es nicht nötig in der Deckliste für alle verfügbaren Decks alle vorhandenen Daten herunterzuladen. Alle Dateioperationen werden in gesonderte Loader ausgelagert, welche alle von der Klasse *AsyncTask* erben und somit parallel zum UI-Thread laufen.

Der *OnlineDecksLoader* fragt lediglich die */decks* Ressource ab um an die IDs der vorhandenen Decks zu kommen über welche dann in einem zweiten Request nach *decks/ID* der Name, Beschreibung und der Pfad für das Coverbild abgefragt werden. Das Coverbild wird anschließend vom gegebenen Pfad heruntergeladen und mittels der Methode *decodeStream()* der Klasse *BitmapFactory* in ein *Bitmap*-Objekt umgewandelt. Dieses wird dann in den Cachepfad (siehe Abbildung 4.1) abgelegt und fortan nichtmehr extra angefragt. Er gibt anschließend eine Liste von fertigen OnlineDecks an seine Listener zurück.

Der *DeckDownloader* kapselt hingegen die ganze Logik des Downloads bis hin zur persistenten Speicherung der gesamten Deckdaten eines über die ID spezifizierten Decks. Er ruft dazu sequentiell hintereinander die verschiedenen Ressourcen-Pfade aus Abschnitt 3.3 ab, ändert nach erfolgreichem Download der Bilder den Bildpfad in den JSON-Objekten und berechnet zusätzlich über allen Attributwerten der Karten jeweils den Median und hängt diesen direkt in die einzelnen Attribute. Anschließend werden alle JSON-Teile zu einem großen JSON-Objekt kombiniert und als File im Dateisystem gespeichert. Zusätzlich wird direkt ein OfflineDeck erstellt und an die Listener zurückgegeben.

4.4 Spiellogik

Wie bereits im Abschnitt 3.1.4 erläutert wird in der *GameActivity* das Spiel ausgeführt. Sie beinhaltet ein Objekt der Klasse *GameEngine*, welche mit Hilfe der an die *Ga-*

meActivity übergebenen Parameter initialisiert wird. Die wichtigen Komponenten der *GameEngine* sind in der Abbildung 4.2 dargestellt.

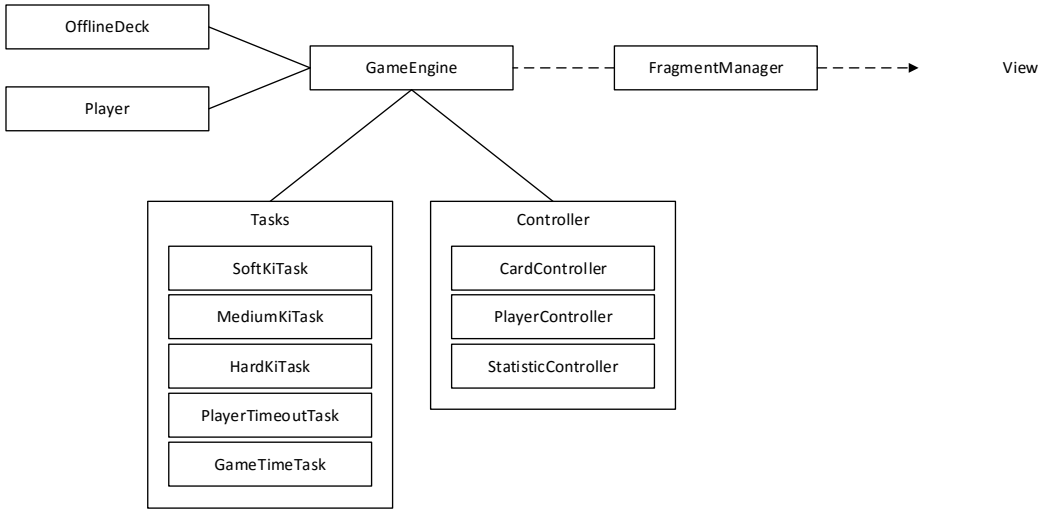

Abbildung 4.2: Aufbau der GameEngine Klasse

Im Folgenden werden die Komponenten aus Abbildung 4.2 kurz erläutert:

- **Player** Jeder Spieler wird in einem Player-Objekt gespeichert.
- **OfflineDeck** Das Kartendeck welches im aktuellen Spiel verwendet wird.
- **Tasks** Hierbei werden verschiedene Typen verwendet, welche von der Klasse *AsyncTask* erben:
  - **SoftKiTask** Einfacher Computergegner, der einen Spielzug ausführt.
  - **MediumKiTask** Mittlerer Computergegner, der einen Spielzug ausführt.
  - **HardKiTask** Schwerer Computergegner, der einen Spielzug ausführt.
  - **PlayerTimeoutTask** Zählt die Zeit herunter, die ein Spieler pro Zug zur Verfügung hat. Falls die Zeit abgelaufen ist wird ein beliebiges Attribut gewählt.
  - **GameTimeTask** Zählt die Zeit herunter, die für ein Spiel festgelegt wurde.
- **Controller** Hierbei wurden verschieden Typen verwendet:

  - **CardController** Regelt die Kartenverteilung zu jedem Zeitpunkt des Spiels.
  - **PlayerController** Regelt die Punkteverteilung und ist für die Festlegung des Gewinners zuständig.
  - **StatisticController** Regelt die Statistiken und schreibt diese in die *Shared Preferences*
- **FragmentManager** Ist dafür zuständig dass nach jeder Runde eine neue Karte angezeigt wird.

Das komplette Spiel beruht auf dem *Listenerkonzept*, d.h. es gibt keine Schleife die dauerhaft nach Ereignissen sucht. Somit ist wird z.B. nach jedem Spielzug oder nachdem ein Dialogfenster geschlossen wurde eine Nachricht an die *GameEnginge* gesendet, die dann weitere Funktionen aufruft.

4.5 Views

Die Views der in Abschnitt 3.1.4 erläuterten Activities sind allesamt als Fragment implementiert. Jede Activity enthält also mindestens ein Fragment, in welchem das eigentliche Layout ausgerollt wird. Rückwirkend war das für dieses Projekt wohl ein unnötiger Zusatzaufwand, da das Game aktuell nur für Smartphones optimiert ist. Der Vorteil alles mit dem Fragment-Konzept zu implementieren hätte sich unter anderem erst ausgewirkt, wenn das Game auch für andere Formfaktoren wie z.B. Tablets hätte optimiert werden müssen, denn dann hätte das Layout mithilfe der Fragments sehr leicht angepasst werden können. Obwohl dies keine direkte Anforderung war wollten wir uns diese Option trotzdem offen halten.

5

Zusammenfassung & Fazit

5.1 Was war schwer?

Bei der Implementation von Android Anwendungen muss man sich daran gewöhnen, dass man das OOP-Konzept wie man es von Java kennt nur eingeschränkt anwenden kann. Oft ist es nicht ohne weiteres möglich oder sollte konzeptionell vermieden werden Objektreferenzen direkt über Konstruktoren oder Mehtoden zu übergeben. Android hat das in Abschnitt 3.1.1 gezeigte Konzept der Lifecycles entworfen um ein intelligentes Speichermanagemant zu erreichen. Das Betriebssystem ist dabei in der Lage die definierten Methoden selbständig aufzurufen und bei Bedarf pausierte Activities im Hintergrund zu schließen. Dabei könnte es bei hart referenzierten Objekten passieren, dass Referenzen automatisch verschwinden wenn Activities im Hintergrund zerstört werden. Deshalb gibt es in Android die Möglichkeit Werte und *serialisierte* Objekte über Intents zwischen den Activities zu übergeben. Das war zu Beginn etwas gewöhnungsbedürftig, da man ständig aufpassen muss, dass alle Klassen die man irgendwo einmal übergeben möchte serialisierbar sind.

Aus ähnlichem Grund und weil Fragments in beliebige Activities eingebettet werden können sollte eine direkte Kommunikation von Fragments zur Parent-Activity über Referenzierung vermieden werden, da die Referenzen dynamisch wechseln können und somit eine Referenz auf eine Parent-Activity auf *null* zeigen könnte. Android nutzt hier und an anderen stellen sehr stark das Listener-Konzept, wobei die Listener bei jedem *onAttach()* Aufruf des Fragments neu gesetzt werden. Auch daran musste man sich im Vergleich zu normalen Java-Programmen wo dies nicht so stark vertreten ist erst einmal gewöhnen.

Durch die vielen Background-Task für Datei- und Netzwerkoperationen oder KI-Tasks wurde sehr viel Komplexität in den Programmablauf integriert, was an einigen Stellen einen sehr hohen Debugging-Aufwand zur Folge hatte.

5.2 Was war leicht?

Bei aller Komplexität ist Android trotzdem relativ angenehm zu implementieren, da die mit Android Studio auf IntelliJ basierende IDE sehr gut mit den Eigenarten von Android umgehen kann und an vielen Stellen grobe Fehler in der Konzeption abfängt. Gerade was Listener angeht macht Android Studio mit Template-activites sehr deutliche Vorschläge wie eine gute Konzeption aussehen könnte.

Sehr effizient ist auch die integrierte Verknüfung der in XML geschriebenen Layout und Views mit dem Java-Code. Durch eine durchgängige Autovervollständigung über die verschiedenen Dateiformate hinweg lässt sich sehr effizient implementieren ohne ständig nach korrekten Bezeichnernamen suchen zu müssen.

Für grobes Konzeptionieren der Layouts eignet sich auch der integrierte Drag-and-Drop Layouteditor sehr gut. Man bekommt in sehr kurzer Zeit ohne überhaupt XML schreiben zu müssen ein ordentliches Layout zusammengeklickt, bei dem nur noch kleinere Anpassungen in XML nötig sind um ein optimiertes Layout zu erhalten.

5.3 Fazit

Bei der Entwicklung einer App müssen viele Besonderheiten der jeweiligen Zielplattformen und der dazu gehörigen Geräte beachtet werden. Um all diese Besonderheiten gut abdecken zu können bedarf es einer guten Planung, durch die viele Fehler vorab beseitigt werden.

Die Implementierung in *Android* war durch die plattformspezifischen Besonderheiten mit einigen Hürden verbunden. Zum Beispiel wurde leider erst im späteren Verlauf der Implementierung erkannt, dass Objekte die an andere *Activities* übergeben werden das

Interface Serializable implementieren müssen. Durch die vorherige Auseinandersetzung mit der *Android API* verlief die Implementierung im Allgemeinen recht gut.

Durch das Anwendungsfach *Mobile Application Lab* schafften wir eine gute Grundlage für die Entwicklung von Apps. Die vorgegebenen Anforderungen der App wurden alle realisiert und darüber hinaus wurden noch eigene Ziele umgesetzt. Durch die Entwicklung sammelten wir viele Erfahrungen und sehen uns jetzt in der Lage selbstständig kleine Apps zu entwickeln.

Insgesamt hat uns das kleine Softwareprojekt viel Spaß gemacht und wir freuen uns auf die kommende Veranstaltung *Mobile Application Development*, in welcher wir unsere eigenen Ideen umsetzen dürfen.

Abbildungsverzeichnis