Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt
Fakultät Informatik und Wirtschaftsinformatik

Bachelorarbeit

Design und Implementierung eines Generators für Android View Komponenten

vorgelegt an der Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt in der Fakultät Informatik und Wirtschaftsinformatik zum Abschluss eines Studiums im Studiengang Informatik

Marcel Groß

Eingereicht am: 31.03.2017

Erstprüfer: Prof. Dr. Peter Braun
Zweitprüfer: Prof. Dr. Steffen Heinzl

Zusammenfassung

TODO

Abstract

TODO

Danksagung

Inhaltsverzeichnis

1 Einführung

Das Smartphone ist heutzugtage der stete Begleiter eines Menschen. „Zwei Drittel der Bevölkerung und nahezu jeder 14- bis 29-Jährige geht darüber ins Netz.“ [5] Auch die Prognose zeigt, das der Absatzmarkt immer weiter steigen wird (Abbildung 1.1).

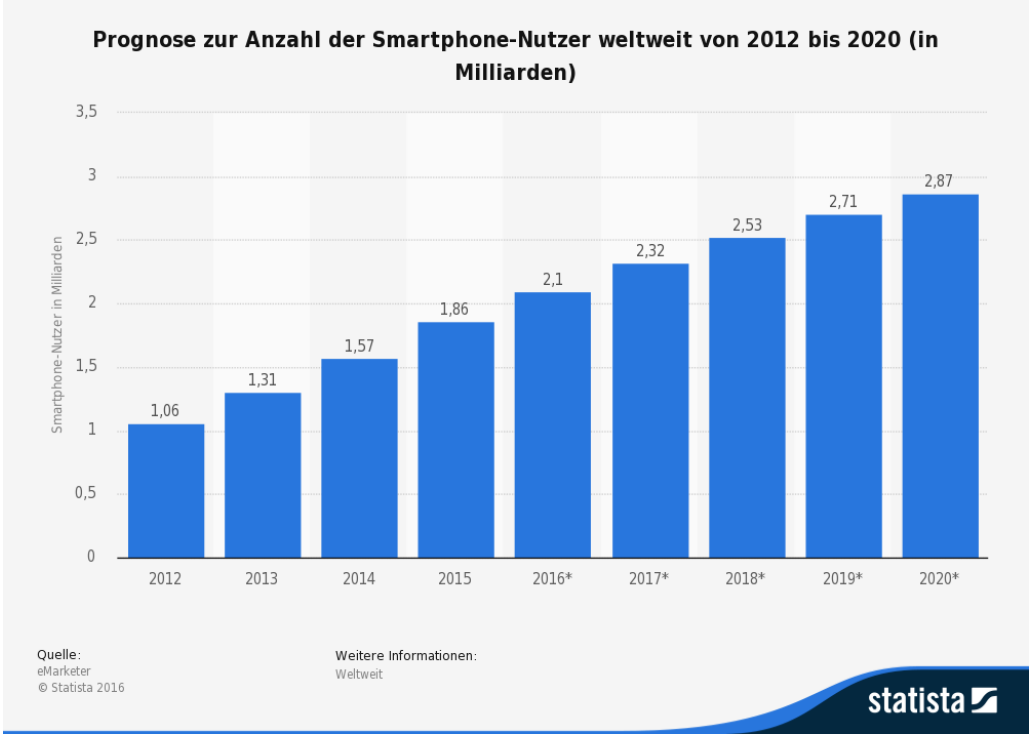

Abbildung 1.1: Prognose zur Anzahl der Smartphone-Nutzer weltweit von 2012 bis 2020 (in Milliarden) [14].

Umso wichtiger ist es das die Softwareentwicklung diesen Trend ernst nimmt. Der ehemalige Google-Chef Eric Schmidt sagte bereits 2010: „Googles Devise heißt jetzt ‚Mobile first‘“. Diese Devise wird von vielen Unternehmen verfolgt, das ist der Grund weswegen in den einzelnen Stores heutzutage so viele Apps angeboten werden. Bei Android im Playstore sind es im Oktober 2016 ca. 2,4 Millionen Apps [3], bei Apple im App Store sind es ca. 2 Millionen Apps (Stand Juni 2016) [13]. Neben Googles Android und Apples

iOs gibt es noch andere Betriebssysteme, beispielsweise Microsofts Windows Phone oder Blackberrys Blackberrys OS. Jedoch bestimmen die beiden erstgenannten Systeme den Markt (Abbildung 1.2).

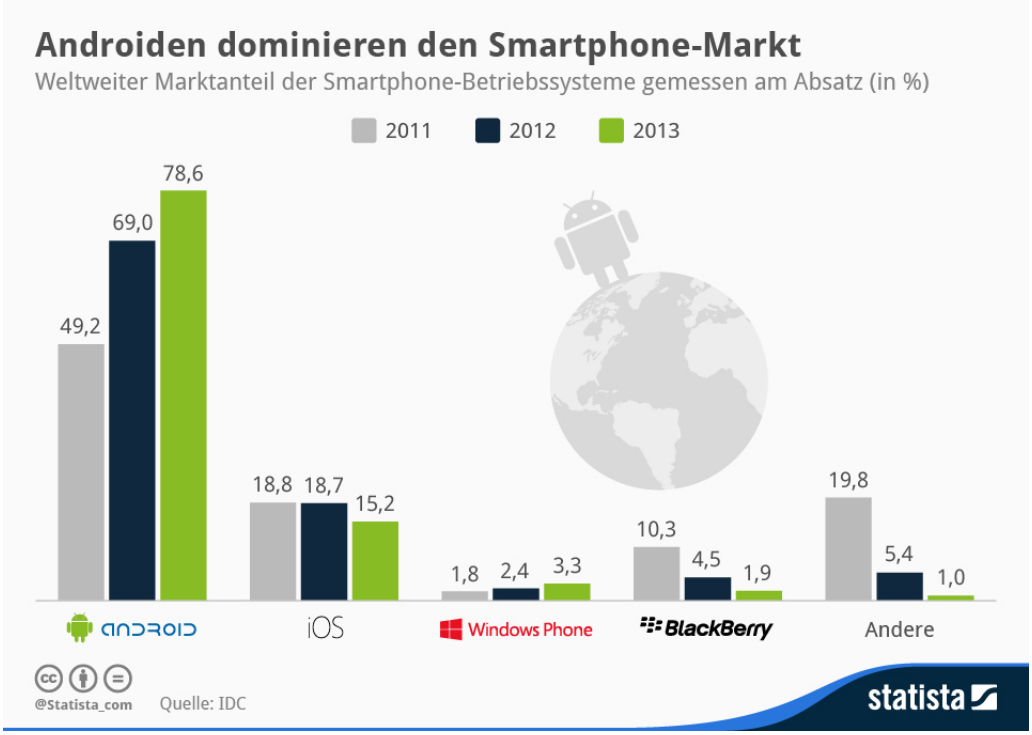

Abbildung 1.2: Der weltweite Marktanteil von Smartphone-Betriebssysteme. [2]

Jede dieser Applikationen wurden einzeln für sich entwickelt und implementiert. Bei jedem Update zum Beispiel des Systems, müssen alle Anwendungen gewartet und überarbeitet werden, um die volle Funktionalität zu gewährleisten.

Würden einige Applikationen jedoch genauer analysiert werden, wäre das Ergebnis, dass in jeder dieser Anwendungen Codepassagen vorhanden sind, welche einen ähnlichen beziehungsweise den selben Zweck erfüllen. Werden diese Stellen im Programmcode abstrahiert, gibt es die Möglichkeit diese generieren zu lassen. Um Code generieren zu lassen, benötigt man so genannte Code-Generatoren.

Im Bereich der Backend-Entwicklung gibt es bereits verschiedene Projekte die sich damit befassen. Ein Beispiel wäre der *CRUD Admin Generator* [7]. Die Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt entwickelt unter der Leitung von Prof. Dr. Peter Braun auch einen Code-Generator unter dem Namen: GeMARA. Mit Hilfe solcher Generatoren für den Bereich von Mobilen Applikationen, könnte der Entwicklungs- und Wartungsaufwand reduziert werden.

Führt ein Systemupdate dazu, dass die Implementierung von verschiedenen Anforderungen nicht weiter funktionsfähig ist, muss dies nur einmalig an der entsprechenden Stelle im Code-Generator geändert werden und nicht in jeder Applikation einzeln.

1.1 Motivation

Im Rahmen des Projektes GeMARA gab es bereits Arbeiten, welche sich mit dem Thema der Generierung von Android Aktivities beschäftigt. Die dabei entstandenen Lösungen, resultieren darin, das dass generieren von Aktivities zu Problemen führt. Deshalb beschäftigt sich diese Ausarbeitung damit, nicht eine komplette Aktivity zu erzeugen, sondern sogenannte Komponenten.

Eine Komponente, ist im wesentlichen eine kleine Anwendung für sich, welche nur eine einzige Aufgabe erfüllt. Dies könnte zum Beispiel das Anzeigen eines Dozenten in einer Campus-Applikation sein.

Aus den erzeugten Komponenten, kann eine Art Bausatz entstehen. Mit dessen Hilfe der Entwickler seine Applikation zusammen bauen kann. Dabei wird ihm freie Wahl gelassen, wie der Aufbau seiner Anwendung aussieht, er bedient sich nur an gegebener Stelle an den Komponenten. Dadurch reduziert sich der Entwicklungsaufwand für ihn.

Bewegen wir uns in der Domain einer Hochschule, so kann eine Bibliothek mit den erzeugten Komponenten allen Studierenden zur Verfügung gestellt werden. Dadurch wäre jeder dieser Studierenden in der Lage eine persönliche Campus-Applikation zu entwickeln. Durch die einzelnen Komponenten kann dann sichergestellt werden, dass grundsätzliche Funktionalität bereits gewährleistet ist.

1.2 Zielsetzung

Ziel dieser Ausarbeitung liegt darin, dass der Leser ein grundsätzliches Verständnis für die Entwicklung von Android-Applikationen beziehungsweise Android-Bibliotheken vermittelt bekommt. Weiterhin soll das Wissen über Datenkommunikation mittels REST vertieft werden. Hierbei wird ein Schwerpunkt auf das Hypermedia-Prinzip gelegt.

Neben diesen spezifischen Anforderungen, soll ein Verständnis für der Implementierung von Generatoren entstehen. Dafür muss der Entwickler entscheiden können, was von der Implementierung als statischer Code angesehen werden kann und welcher generisch ist.

Dieses Verständnis ist wichtig, um die Komplexität der Generatoren zu reduzieren. Da die statischen Anteile jedes mal identisch sind.

Auch soll auf die Frage eingegangen werden, ob man das User Interface (UI), welches ebenfalls generiert wird, auch generisch gestalten kann. Das bedeutet, dass nicht nur die Informationen, welche angezeigt werden sollen beschreibt. Sondern auch wie diese angezeigt werden sollen.

Wenn es möglich ist dass das UI als Teil der DSL beschrieben werden kann, so hat der Nutzer des entsprechenden Generators die Freiheit, selbst zu entscheiden ob zum Beispiel bei seiner Campus-App, bei der Liste aller Dozenten das Profilbild links oder rechts angezeigt werden soll.

1.3 Aufbau der Arbeit

Aufbau

2 Grundlagen

2.1 REpresentational State Transfer (REST)

In dem generierten Projekt, sollen alle benötigten Daten mittels REST von dem zugehörigen, generierten Backend geladen werden.

REST [4] ist ein Programmierparadigma, welches sich auf folgende Prinzipien stützt: Client-Server, Zustandslose Kommunikation, Caching, Uniform Inferface Layered System und dem optionalen Prinzip Code-on-Demand.

Diese Arbeit berücksichtigt vor allem Hypermedia as the Engine of Application State (HATEOAS) welches unter das Prinzip Uniform Interface fällt. Es beschreibt, wie mit Hilfe eines endlichen Automaten eine REST-Architektur entworfen werden kann.

Der Architekt einer REST-konformen Application Programming Interface (API) überlegt sich im voraus, wie der Applikation-Fluss in der späteren Anwendung aussehen soll. Dafür definiert er verschiedene States und welche Transitionen zum nächsten State führen.

Als ein State kann beispielsweise das Anzeigen aller Lecturer in einer Campus-Applikation angesehen werden. Die Transition hingegen ist zum Beispiel ein Link im Link-Header der Antwort, oder ein Attribut, der empfangen Ressource.

Wird das API mit Hilfe eines endlichen Automaten entwickelt, kann diese dem Client-Entwickler als Anleitung zum erstellen seines Clients dienen. Er benötigt lediglich einen Uniform Resource Locator (URL), welcher auf den initialen State des endlichen Automaten führt. Dieser liefert dann alle, zu diesem Zeitpunkt möglichen, Transitionen zurück. Mit Hilfe dieser Transitionen, kann sich der Entwickler dann zum nächsten State bewegen. Auch dieser State liefert neben den Ressourcen, alle möglichen weiteren Transaktionen zurück. Wenn der Entwickler sich so durch die States bewegt, bekommt er die benötigten Informationen zum Aufbau und Ablauf der Applikation.

Die Abbildung 2.1 zeigt einen solchen Automaten. Der Einstiegspunkt ist der State „Dispatcher“ dieser liefert die Tansition zum State „Collection“ zurück. Dieser State,

verfügt über alle Informationen die benötigt werden um eine Collection der betroffenen Ressource anzuzeigen, weiterhin verfügt er auch das Wissen, über die beiden nächsten Transitionen zu den States „Create“ und „Single“. Wie der Name des States annehmen lässt, wird der State „Create“ benötigt um eine neue Ressource anzulegen. Von diesem State aus kann die Anwendung nur zurück zum State „Collection“. Der State „Single“ enthält alle benötigten Daten um eine einzelne Ressource anzuzeigen. Vom hier kann die Anwendung zum State „Update“ oder „Delete“ wechseln. Der State „Update“ ermöglicht es die Ressource zu bearbeiten. Von hier kann der Nutzer der Anwendung nur zum State „Single“ zurückkehren. Der State „Delete“ löscht die aktuelle Ressource und liefert die Transition zum State „Collection“ zurück. Dieses Beispiel verdeutlicht noch einmal bildlich, das der Entwickler nur den Einstiegspunkt „Dispatcher“ kennen muss. Die Anwendung liefert selbst alle benötigten Informationen um die Daten für die Anwendung nachzuladen.

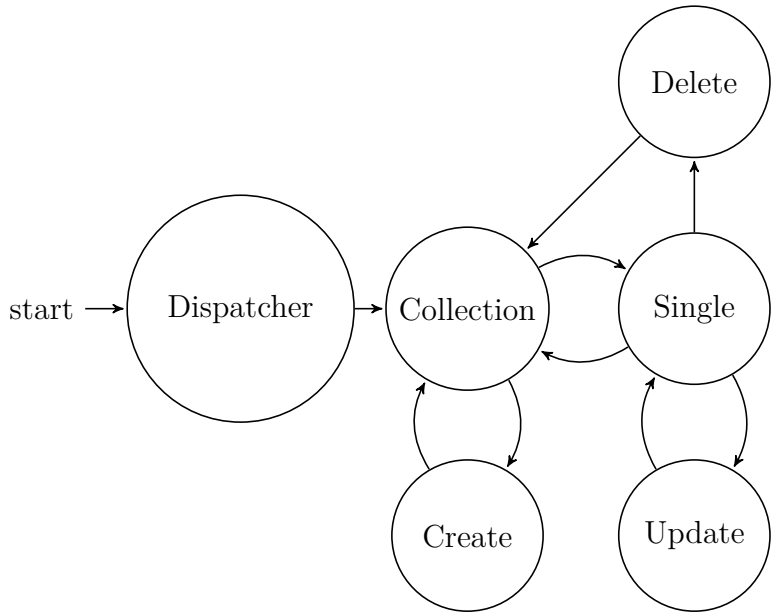

Abbildung 2.1: Aufbau eines REST-API mit Hilfe eines endlichen Automaten.

2.2 Entwicklung von Android CustomViews

Die Software-Plattform Android basiert auf Linux und wird als Betriebssystem für mobile Endgeräte verwendet. Das System wird als Open Source Projekt von der Open Handset Alliance entwickelt [12]. Dabei ist ein Ziel, die Schaffung eines offenen Standards für mobile Endgeräte. Die Entwicklung ist nicht abgeschlossen, die aktuelle Version ist 7.0 Nougat (Stand Feb. 2017).

Programme für diese Plattform nennt man Applikation oder kurz App. Eine App stellt alle nötigen Sourcen bereit, zum Beispiel den Programmcode, Layout und Grafiken, die benötigt werden, um diese App auf einem Android-Endgerät auszuführen.

Mit Hilfe von Widgets und Layouts können Views definiert werden. Diese Views stellen dann die gewünschte Information auf dem Display dar. Die bekanntesten Widgets sind: TextView, Button und EditText. Die Anordnung dieser Widgets erfolgt dann mit einem Layout. Es gibt hierbei verschiendene Layouts zur Auswahl. Beispielsweise das LinearLayout, mit horizontaler oder vertikaler Orientierung. Ein weiteres Beispiel ist das RelativeLayout.

Reichen die Standart-Layouts und -Widgets nicht aus, gibt es noch die Möglichkeit eigene zu entwickeln. Dies ermöglicht diese Views um Attribute und Methoden zu erweitern. Diese können dann sowohl in der Layout-XML als auch im Programm-Code angesprochen werden.

Ausgehend davon das eine Applikation eine Liste von Personen mit Hilfe einer RecyclerView anzeigen soll, gibt es die Möglichkeit eine CardView zu erzeugen, welche eine einzelne Person darstellt. Diese CardView kann in einem XML-Layout wie allgemein bekannt definiert werden. Um die View dann mit den entsprechenden Informationen zu befüllen werden im Adapter der ListView dann die einzelnen Attribute einzeln angesprochen und mit den erforderlichen Details befüllt.

Alternativ besteht die Möglichkeit eine CustomView zu erzeugen, in diesem Fall eine PersonCardView. Hierfür sind folgende Schritte notwendig: Registrieren der CustomViews, Definieren des Aufbaus der CustonView erzeugen einer CustomView Klasse.

2.2.1 Registrieren der CustomView

Zur Erzeugung und Registrierung von CustomViews wird eine Datei „attrs.xml“ benötigt. Diese wird liegt im Ordner „values“ im Verzeichnis „res“. In dieser XML-Datei werden im „resoruces“-Bereich die einzelnen CustomViews aufgelistet. Es besteht die Möglichkeit diesen Views zusätzlich Attribute zuzuweisen. Ein Attribut besteht dabei immer aus einem Namen und einem Format. Dieses Format definiert den erwarteten Eingabewert. Es gibt folgende definierte Formate: string, integer, boolean oder color. Formate können kombiniert werden. Beispielsweise das Attribut „backgroundColor“ könnte so definiert werden format=„color|string“. Listing 2.1 zeigt den Aufbau einer „attrs.xml“-Datei.

Listing 2.1: Aufbau einer „attrs.xml“ - Datei

```
<resources>
    <declare-styleable name="AttributeInput">
        <attr name="hintText" format="integer"/>
        <attr name="inputType" format="string"/>
    </declare-styleable>

    <declare-styleable name="PersonCardView" />
</resources>
```

2.2.2 Definieren des Aufbaus der CustomView

Da die PersonCardView eine CustomView ist, welche aus verschiedenen Widgets zusammengesetzt wurde, müssen diese auch definiert werden. Dies geschieht wie gewohnt mit Hilfe einer XML-Datei, mit einer Ausnahme. Die Root-View ist in diesem Fall keine CardView sondern ein beliebiges anderes Layout. Da die PersonCardView von CardView erbt und somit bereits eine CardView ist.

Listing 2.2: Aufbau der PersonCardView mit Hilfe einer XML-Datei

```
<RelativeLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/relativeLayout"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <TextView
        android:id="@+id/first_name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/firstName"/>

    <TextView
        android:id="@+id/last_name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/last_name"/>

...

</RelativeLayout>
```

2.2.3 Erzeugen einer CustomView Klasse

Hierfür wird eine neue Java-Klasse erzeugt, welche von CardView erbt. Es kann auch direkt von der View-Klasse geerbt werden und anschließend mithilfe der Methode „onDraw“, welche überschrieben werden muss, den gewünschten Inhalt anzuzeigen Sei es nun Text, Formen oder Benutzereingaben. In diesem Fall entspricht die CardView weitestgehend bereits den Anforderungen, so das diese genutzt wird. Die Vererbungsstruktur bringt mit sich, das die Konstruktoren der CardView implementiert werden müssen. Die Anzahl dieser Konstruktoren hängt von der Minimum SDK-Versions des Projekts ab. Dieses Projekt nutzt das Minimum Level 12 somit müssen drei Konstrukotoren überschrieben werden. Ab einen Level von 21, sind es vier, da ein weiteres Attribut zur View hinzugefügt wurde.

Listing 2.3: Konstruktoren der PersonCardView

```
1  public class PersonCardView extends CardView {
2
3  public PersonCardView(Context context) {
4      super(context);
5      init(context, null, 0);
6  }
7
8  public PersonCardView(Context context, AttributeSet attrs) {
9      super(context, attrs);
10     init(context, attrs, 0);
11 }
12
13 public PersonCardView(Context context, AttributeSet attrs, int
       defStyleAttr) {
14     super(context, attrs, defStyleAttr);
15     init(context, attrs, defStyleAttr);
16 }
17 ...
18 }
```

Innerhalb der „init“-Methode wird definiert, was die View anzeigen beziehungsweise was sie tun soll. In diesem Beispiel werden die verwendeten Widgets initialisiert. Besäße die PersonCardView noch eigene Attribute, so würden diese im AttributeSet übergeben und könnten daraus in ein TypedArray geschrieben werden. Dieses TypedArray muss am Ende „recycled“ werden, damit es für einen späteren Aufruf wieder zur Verfügung steht.

Jetzt wird die PersonCardView um eine Methode „setPerson“ erweitert. Diese ist angelehnt an die Methode „setText“ der TextView. Sie ermöglicht das der PersonCardView ein Objekt Person übergeben wird und füllt dann die entsprechenden Widgets mit den dazugehörigen übergebenen Daten.

Listing 2.4: *setPerson* - Methode aus der PersonCardView.

```
1 public void setPerson(Person person) {
2     this.firstName.setText(person.getFirstName());
3     this.lastName.setText(person.getLastName());
4     ...
5 }
```

Mit Hilfe dieser Methode wird die Nutztung der PersonCardView vereinfacht. Im Adapter der RecyclerView wird jetzt nicht mehr jedes einzelne Widget definiert und mit Informationen befüllt. Sondern nur noch die PersonCardView und mit der „setPerson“ - Methode kann die komplette Karte mit den Daten einer Person mit nur einem Methodenaufruf befüllt werden.

2.3 Software-Generatoren

Mit Software-Generatoren, ist es möglich Software generieren zu lassen. Dafür wird die Problemstellung der realen Welt so beschrieben, dass der Generator dies versteht, interpretieren und Programmcode erzeugen kann.

2.3.1 domänenspezifische Sprache (DSL)

Die Grundlage, um ein Model für einen Generator zu beschreiben ist die domänenspezifische Sprache. Eine DSL ist eine Programmiersprache, welche auf die Probleme einer bestimmten Domäne ausgelegt ist [9]. Dadurch dass diese Sprachen auf ein ganz bestimmtes Problem zugeschnitten sind, sind domänenspezifische Sprache in ihrer Ausdrucksfähigkeit beschränkter als herkömmliche Programmiersprachen wie beispielsweise Java, C++ oder C# Eine domänenspezifische Sprache wird dafür entwickelt ein konkretes Problem so effizient wie möglich zu lösen, ohne die komplexen Strukturen des Programmcodes kennen zu müssen.

Bekannte Domänenspezifische Sprachen sind: Structured Query Language (SQL), Make und HyperText Markup Language (HTML).

Die domänenspezifischen Sprachen lassen sich in zwei Kategorien einteilen die internen und die externen DSLs.

Interne DSLs

Eine interne DSL wird auch embdded DSL genannt, weil sie keine eigene Syntax und Grammatik entwickelt. Sie bedienen sich der Hostsprache. Das heißt sie nutzen die selbe Programmiersprache, in welcher auch das Resultat sein wird.

Jedoch wird die verwendete Hostsprache eingeschränkt, so nutzt die domänenspezifische Sprache nur eine Teilmenge der Möglichkeiten [8]. Die Nutzung von internen DSLs sind zum Beispiel: Es muss kein neuer Compiler und Parser geschrieben werden. Auch gibt es bereits integrierte Entwicklungsumgebungen (IDEs). Außerdem muss der Programmierer keine neue Sprache lernen, um die domänenspezifische Sprache zu nutzen, sollte er die verwendete Hostsprache bereits kennen.

Externe DSLs

Anders als die internen DSLs besitzen die externen DSLs eine eigene Syntax. Dies macht die Entwicklung einer solchen domänenspezifische Sprache sehr viel aufwändiger, da nun ein eigener Parser und Compiler mitentwickelt werden muss [8]. Jedoch bringt diese eigene Syntax auch den Vorteil, dass die Sprache nicht auf die Besonderheiten einer Hostsprache eingeschränkt ist. So können Anforderungen an die Domäne bereits beim schreiben des Compilers mit validiert werden.

2.3.2 GEnerierung von Mobilen Applikationen basierend auf REST Architekturen (GeMARA)

Das Projekt GeMARA beinhaltet ein Reihe von Software-Generatoren, deren Ziel es ist mobile und verteilte Applikationen basierent auf dem REST Architekturstil generieren zu lassen. Dafür wurde eine interne DSL entwickelt, mit dessen Hilfe sowohl clientseitige als auch serverseitige Applikationen beschrieben werden können.

Im Moment (Stand Februar 2017) ist es möglich ein WAR-Artefakt für einen Tomcat-Webserver generieren zu lassen. Dieses erzeugte Projekt kann auf eine relationale MYSQL-Datenbank oder einer dokumentenbasierten CouchDB zugreifen. Des weiteren ist ein Generator zur Erzeugung von Android-Applikationen sowie ein Generator für Polymer Webkomponenten in der Entwicklung.

Aufbau von GeMARA

GeMARA ist modular aufgebaut, jedes der einzelnen Module erfüllt einen eindeutig definierten Zweck.

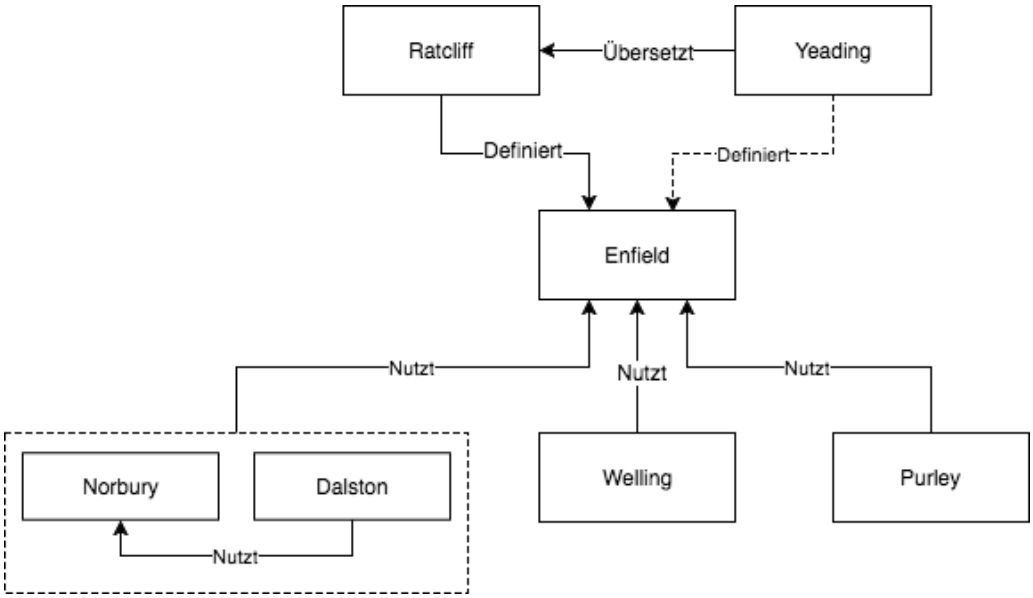

Abbildung 2.2: Aufbau von GeMARA

- **Ratcliff** definiert ein Enfield-Model, mit Hilfe einer Fluent API.
- **Yeading** definiert eine Repräsentation eines Enfield-Models, mit YAML Ain't Markup Language (YAML) oder JavaScript Object Notation (JSON), welches dann in nach Ratcliff übersetzt werden.
- **Enfield** liefert das Meta-Model, welches für die Beschreibung der gewünschten Appliktion benötigt wird.
- **Norbury** stellt für Server-seitige Applikationen den Plattform Code bereit.
- **Dalston** ist ein Software-Generator, welcher den Server-seitigen Code in Java generiert.
- **Welling** ist ein Software-Generator, welcher Android Applikationen generiert.
- **Purley** ist ein Software-Generator, welcher Polymer Webkomponenten generiert.

Diese Arbeit behandelt das Design und die Entwicklung von **Welling**.

3 Problemstellung

3.1 Meta-Model und Android Applikation spezifisches

Bei der Generierung von Android Applikations gibt es vieles zu berücksichtigen. Angefangen bei den einfachsten Möglichkeiten zur Darstellung von Schrift. In welcher Farbe oder Größe sollte sie dargestellt werden.

Wie soll eine View an sich aufgebaut sein? Wie sind die zu repräsentierenden Daten aufbereitet? Soll der Vorname eine Zeile über dem Nachnamen stehen? Oder soll es genau umgekehrt sein? Es gibt auch noch die Möglichkeit der Kombination. Vorname und Nachname in einer Zeile, oder Nachname und Vorname in einer Zeile.

Was soll überhaupt dargestellt werden? Gehen wir vom Beispiel Person aus, soll nur der komplette Name dargestellt werden oder nur ein Teil des Namens. Was ist mit dem Geburtstag oder dem Wohnort. Gibt es zu der Person ein Profilbild? Was passiert wenn nicht alle Personen ein Profilbild haben, aber es soll ein Profilbild angezeigt werden? Das sind ein Teil der Fragen, die sich rein auf das User Interface (UI) beziehen. Es gibt aber noch weiter Fragen die gestellt werden müssen. Soll es die Möglichkeit geben, das Aktionen beim Klick auf die Telefonnummer, E-Mail oder Homepage einer Person klickt ausgefüht werden?

Oder noch elementarer, welche Ansichten soll es überhaupt geben? Listen von Personen, Detailansichten und so weiter. Soll es die Möglichkeit geben neue Personen anzulegen, wenn ja was sind Pflichtangaben zu einer Person? Dürfen bestehende Personen bearbeiten werden können?

Die letzten Fragen bezogen sich auf mögliche Funktionalitäten der Anwendung. Im letzten Bereich, gibt es noch Fragen, bezüglich des Ablaufes in einer Applikation. Welche View kommt nach welcher Aktion, wie sieht der Flow innerhalb einer Applikation aus.

Die oben genannten Fragen sind nur Beispiele für Überlegungen welche betrieben werden müssen um eine Android Anwendung zu entwickeln, das unterscheidet sich nicht vom normalen Entwicklungsprozess einer Anwendung. Die große Frage hinter den aufgezählten Problemstellungen ist, wie können diese Anforderungen soweit abstrahiert werden, das diese Möglichst einfach mit einer domänenspezifische Sprache (DSL) beschrieben werden können.

Wurde ein geeignetes Meta-Model gefunden, so bleibt noch der Aspekt, dass der Software-Generator als Modul von GEnerierung von Mobilen Applikationen basierend auf REST Architekturen (GeMARA) entwickelt werden soll. GeMARA bringt ein bereits bestehende Meta-Model und wiederum eigene Anforderungen mit sich. Durch den auf REpresentational State Transfer (REST), basierenden Architekturstil bringt es beispielsweise die Anforderung mit, dass eine Anwendung mit Hilfe eines endlichen Automaten designed werden soll. Das bedeutet, dass die States und Transitionen des endlichen Automaten den Ablauf in der Applikaition vorgeben. Es wäre außerdem noch wünschenswert, dass die Erweiterung des Meta-Models nicht ausschließlich für Android Applikationen, sondern für jegliche Client Anwendungen genutzt werden kann.

Aus den oben erörterten Fragen und Problemstellungen, welche nur beispielhaft und nicht komplett sind, lassen sich nun folgende Kategorien ableiten: die Beschreibung des User Interface, die Beschreibung der Aktionen bei Klick, die Beschreibung der Architektur und des Ablaufs innerhalb der Applikation und die Kompatibilität mit GeMARA und andern möglichen Clients.

Diese müssen für das Design und die Entwicklung eines Software-Generators für Android Applikationen Beachtung finden.

3.2 Design des Software-Generators

Dieses Kapitel zeigt Probleme und Fragestellungen rund um den Generator an sich auf. Selbst wenn ein ein geeignetes Meta-Model besteht, heißt dass noch nicht, dass es auch einen funktionierenden Software-Generator gibt. Es gibt noch zu viele offene Punkte, wobei der Einfachste lautet: Muss alles generiert werden, oder gibt es Dateien, welche kopiert werden können? Macht es Sinn, die Android Anwendung vorher soweit zu abstrahieren, das es möglichst wenig spezifischen Code und viel generischen Code gibt? Wie wird der Generator gesteuert, können die Dateien einfach so generiert werden, oder gibt es Abhängigkeiten untereinander? Was gibt es beim generieren von Klassen zu beachten? Beispielsweise müssen Activies in der *AndroidManifest.xml* registriert werden oder Strings sollten in einer *strings.xml* stehen und im Programmcode sollte nur mit Ids darauf referenziert werden.

Der Ablauf, wie wann was generiert wird muss teilweise im Meta-Model und teilweise im Generator selbst festgelegt werden. Da stellt sich wieder die Frage, was wird wo geregelt?

Für Android Applikationen werden die verschiedensten Arten von Dateien benötigt. Angefangen mit Java-Klassen und XML-Dateien über Gradle-Dateien und Java Archiven (JARs). Das wiederum wirft die Frage auf wie können die einzelnen Datei-Typen generiert und oder kopiert werden?

4 Lösung

Dieses Kapitel befasst sich mit den Möglichkeiten und Lösungsansätze, zu den Problemstellungen aus Kapitel 3. Anhand von Beispielen wird verdeutlicht, wie gewisse Anforderungen umgesetzt werden könnten und wurden.

4.1 Meta-Model

Nach der Anforderungsanalyse, wurden alle relevanten Informationen erkannt und zusammen gestellt. Diese Zusammenstellung an Daten, welche die Applikation beschreiben wird Meta-Model genannt.

4.1.1 Kompatibilität mit GeMARA und andern möglichen Clients

Um die Kompatibilität mit GEnerierung von Mobilen Applikationen basierend auf REST Architekturen (GeMARA) zu waren, wurde das Enfield-Meta-Model untersucht. Mit Hilfe dieser Untersuchung konnte festgestellt werden, an welcher Stelle zusätzliche Informationen für die Clients am sinnvollsten eingebaut werden können. So das diese den Ablauf der Applikation gut beschreiben und an den benötigten Stellen alle relevanten Informationen für den Software-Generator zur Verfügung stellt.

Die Abbildung 4.1 zeigt die vereinfachte Model-Klasse des Enfield-Meta-Models. In dieser Klasse sind bereits die wichtigsten Informationen wie zum Beispiel der Name der Applikation, oder unter welchem Package diese zu finden ist. Neben diesen grundsätzlichen Informationen liefert die Model-Klasse auch den Startpunkt des endlichen Automaten, welcher die Anwendung beschreibt. Dieser Startpunkt ist der *GetDispatcherState*. Dieses Objekt besitzt das Attribut *transitions*. Dieses Attribut beschreibt, welche States auf den Dispatcher-State folgen können. Jeder dieser folgenden States, besitzt wiederum eine Collection mit Transitionen, die auf die nachfolgenden States verweisen. So wird mit Hilfe der Transitionen und der States der endliche Automat der Anwendung beschrieben. Der Generator kann diese Beschreibung nutzen, um zu entscheiden in welcher Reihenfolge, welche Klassen generiert werden müssen.

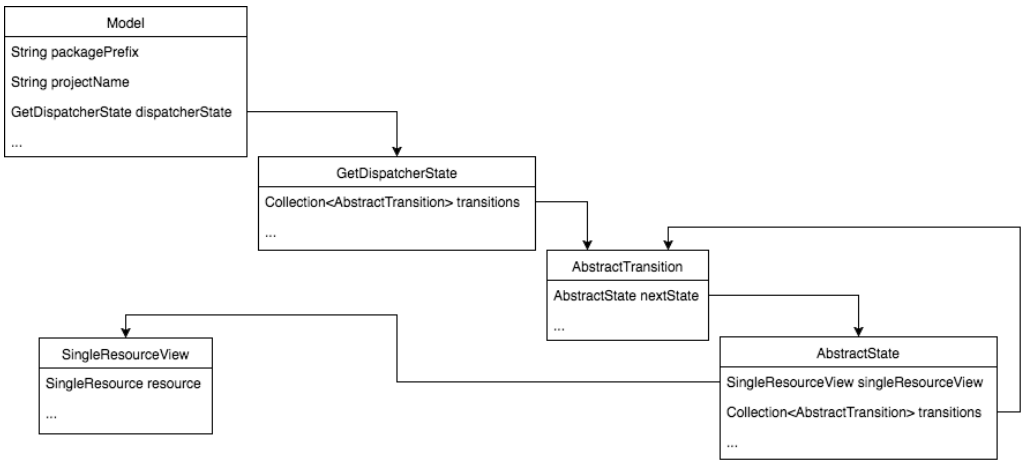

Abbildung 4.1: Vereinfachter Aufbau des Enfield-Meta-Models

Um jetzt zusätzlich benötigten Informationen für die Android Applikation in dieses bestehende Modell einzubauen, gibt es zwei Möglichkeiten.

4.1.2 Eigenes Android-Meta-Model

Es besteht die Möglichkeit die Model-Klasse um ein Attribut *Android-Meta-Model* zu erweitern. Die Abbildung 4.2 stellt zeigt schemenhaft ein Beispiel wie ein Android-Meta-Model aussehen könnte. Auffällig hierbei ist das viele Informationen, die das Enfield-Model bereits liefern würde, hier noch einmal explizit beschrieben werden muss. Ein Beispiel wären die Transitionen, zwischen den Fragmenten beziehungsweise zwischen den Activities.

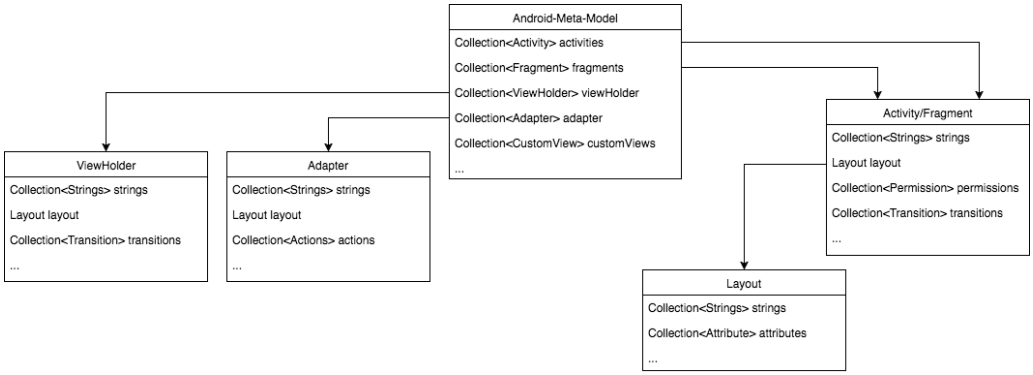

Abbildung 4.2: Möglicher Aufbau eines Android-Meta-Models

Der Nutzer des Software-Generators, muss also ziemlich viel über den Ablauf und die Funktionsweiße einer Android-Anwendung wissen, um diesen Generator sinnvoll verwenden zu können. Dabei bleibt zusätzlich noch die Möglichkeit, das der Nutzer eigens geschriebene Metohden in das Model einpflegen kann. John Abou-Jaoudeh at al., haben in ihrer Arbeit *A High-Level Modeling Language for Efficent Design, Implementation, and Testing of Android Applications* ein Meta-Model entwickelt, welches genau solche Features unterstützt [1].

Der Vorteil einer solchen Erweiterung des Enfield-Models ist, das alle benötigten Daten für die Android Anwendung an einer Stelle zu finden sind. Auch hat der Nutzer die Möglichkeit an manchen Stellen eigene Methoden einzufügen und somit ist er in der Lage das Verhalten der App weiter zu individualisieren.

Jedoch überwiegen in diesem Fall die Nachteile. Ein Nachteil dieses Vorgehens ist, die redundante Beschreibung des Programm-Ablaufes. Einmal im Android-Meta-Model und einmal im Enfield-Meta-Model. Bei jeder Änderung gilt dies zu berücksichtigen. Der nächste Nachteil ist der Nutzer des muss sich in der Entwicklung von Android Anwendungen auskennen. Er muss genau das Zusammenspiel von ViewHoldern, Adaptern, Fragments und Activities kennen. Er muss wissen wie diese ineinandergreifen und wann welche Aktionen ausgelöst werden müssen. Weiterhin sollte er ein Grundsätzliches Verständnis für das Model View Controller (MVC) Pattern besitzen, welches bei der Entwicklung von Android Applikationen anwendung findet. Ein weiterer Nachteil ist die Beschränkung des Models auf Android. Wird das Enfield-Model um ein Android-Meta-Model erweitert, so muss dieses für jeden einzelnen Client geschehen. Soll der Generator beispielsweise um Polymer-Webkomponente oder einer iOS-Anwendung erweitert werden, so müsste für jede einzelne Art von Client, das Enfield-Model mit einem Entsprechenden Meta-Model erweitert werden.

4.1.3 Allgemeine Erweiterungen des Enfield-Models an entsprechender Stelle

In dieser Arbeit wurde sich für die Variante entschieden, das Enfield-Model an ihren *SingleResourceViews* zu erweitern. Wird diese Klasse um die Attribute, die benötigt werden Attribute erweitert, so erhält der Generator die benötigten Ressourcen immer zur rechten Zeit.

Wird beispielsweise eine Instanz eines *GetPrimarySingleResourceByIdStates* erzeugt, und dessen *SingleResourceView* enthält alle notwendigen Informationen, um die View in der Android Anwendung zu beschreiben. Kann der Generator mit Hilfe der Transitionen über die States iterieren und verfügt an jedem State über alle benötigten Informationen, um den aktuellen State in der Anwendung generieren zu lassen.

Bei dieser Methode befinden sich alle State-spezifischen Daten direkt am State. Jedoch gibt es neben diesen spezifischen Daten auch Daten, welche die komplette Applikation betreffen, muss das Enfield-Model noch an einer andern Stelle erweitert werden. Hierfür erscheint es sinnvoll die Erweiterung direkt in der Model-Klasse vorzunehmen. So kann der Generator schon am Anfang auf diese Daten zugreifen und diese verarbeiten.

Die Abbildung 4.3 zeigt das Enfield-Model, welches um die oben genannten Informationen erweitert wurden.

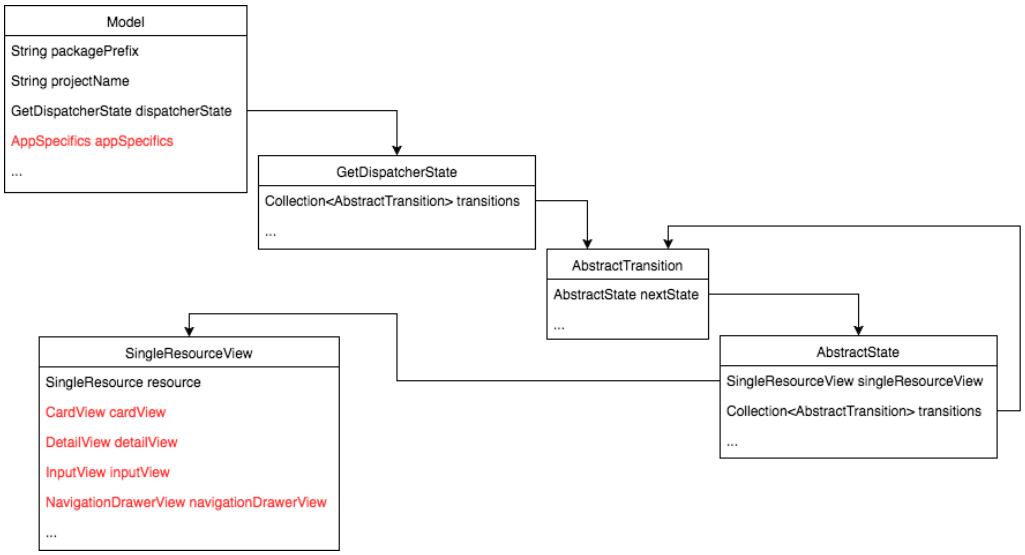

Abbildung 4.3: Vereinfachter Aufbau des erweiterten Enfield-Meta-Models

Der Nachteil dieser Methode ist, das die Informationen an mehr als einer Stelle im Enfield-Modell zu finden sind. Sollten die Informationen zu den Clients verändert werden, so sind Änderungen an der SingleResourceView-Klasse und in der Model-Klasse nötig. Die Vorteile wurden jedoch oben schon einmal erwähnt. Der Generator kann das Model als Fahrplan nutzen und weiß genau wann er welche Klassen für die Android Anwendung erzeugen muss. Er kann auch mit Hilfe der Transitionen bestimmen wie der Verlauf innerhalb der Anwendung ablaufen soll.

4.1.4 Analyse der benötigten Dateien für das Meta-Model

Nachdem identifiziert wurde, an welchen Stellen das Enfield-Model erweitert werden soll, muss noch analysiert werden, welche Informationen an diesen Stellen zur Verfügung stehen müssen. Bei dieser Analyse muss auch ein Augenmerk darauf gelegt werden, wie man die Informationen so aufbereitet, dass diese nicht nur eine Android-Applikation, sondern mögliche andere Clients unterstützen.

Die Analyse in dieser Arbeit beschränken sich auf die Clients Android und Polymere-Webkomponente. Bei beiden wird das User Interface (UI) nach den Guidelines,des von Google entwickelten Material Design, erstellt [11]. Diese Guidelines schreiben bereits viele nötigen Informationen für die Oberflächengestaltung vor. So wird beispielsweise definiert, das Einträge in einer Liste, als Karte dargestellt werden sollen. Abstände und Icons werden ebenfalls festgelegt.

CardView

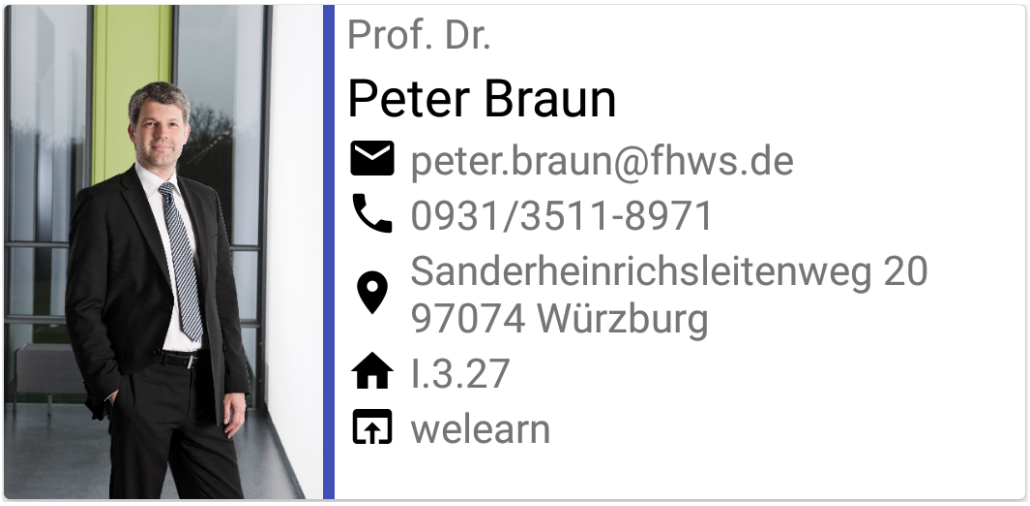

Abbildung 4.4: Beispiel einer CardView aus einer Liste von Dozenten nach Material Design Guidelines.

Listing 4.1: Demo Daten eines Dozenten.

```
...
{
    "address": "Sanderheinrichsleitenweg 20 97074 Wuerzburg",
    "chargeUrl": {
        "href":
   "https://apistaging.fiw.fhws.de/mig/api/lecturers/4/charges",
        "rel": "chargeUrl",
        "type": "application/vnd.fhws-charge.default+json"
    },
    "email": "peter.braun@fhws.de",
    "firstName": "Peter",
    "homepage": {
        "href": "http://www.welearn.de/.../prof-dr-peter-braun.html",
        "rel": "homepage",
        "type": "text/html"
    },
    "id": 4,
    "lastName": "Braun",
    "phone": "0931/3511-8971",
    "profileImageUrl": {
        "href":"https://apistaging.fiw.fhws.de/.../4/profileimage",
        "rel": "profileImageUrl",
        "type": "image/png"
    },
    "roomNumber": "I.3.27",
    "self": {
        "href": "https://apistaging.fiw.fhws.de/mig/api/lecturers/4",
        "rel": "self",
        "type": "application/vnd.fhws-lecturer.default+json"
    },
    "title": "Prof. Dr."
}
...
```

Die JavaScript Object Notation (JSON) Repräsentation unter Listing 4.1 beschreibt das Beispiel aus Abbildung 4.4. Jetzt gilt es zu überlegen, wie die Attribute des JSON Objekts aufzubereiten sind, dass diese die Karte des Dozenten widerspiegeln.

In erster Linie muss entschieden werden, welche der gelieferten Informationen sollen in der Liste für jeden einzelnen Dozenten angezeigt werden. Ist es sinnvoll ist Informationen zu gruppieren? Hier beispielsweise die Attribute *firstName* und *lastName*, diese sollen in einer Zeile angezeigt werden. Ist bekannt welche Informationen eine Karte enthalten

soll, so muss auch noch die Reihenfolge der einzelnen Attribute der Karte bestimmt werden. Neben der Reihenfolge gibt es noch die Möglichkeit das die Schriftgröße oder die Schriftfarbe der einzelnen Attribute unterschiedlich sein können. Auch müssen die Standardicons den einzelnen Attribute zuweisen werden. Auch sollte es die möglich sein einzelnen Attribute bestimmte Aktionen zuzuweisen. So sollte beispielsweise beim Klick auf eine Homepage auch diese im Browser geöffnet werden, oder beim Klick auf die Adresse sollte die Applikation Maps öffnen und die angeklickte Adresse dort anzeigen. Gibt es ein Attribut mit dem Hyperlink zu einer Website, sollte es möglich sein einen Text anzugeben, der anstelle des Hyperlinks angezeigt wird.

Besitzt die Karte ein Bild, so sollte der Nutzer die Möglichkeit besitzen zu entscheiden ob er dieses gerne auf der linken oder der rechten Seite der Karte haben möchte.

DetailView

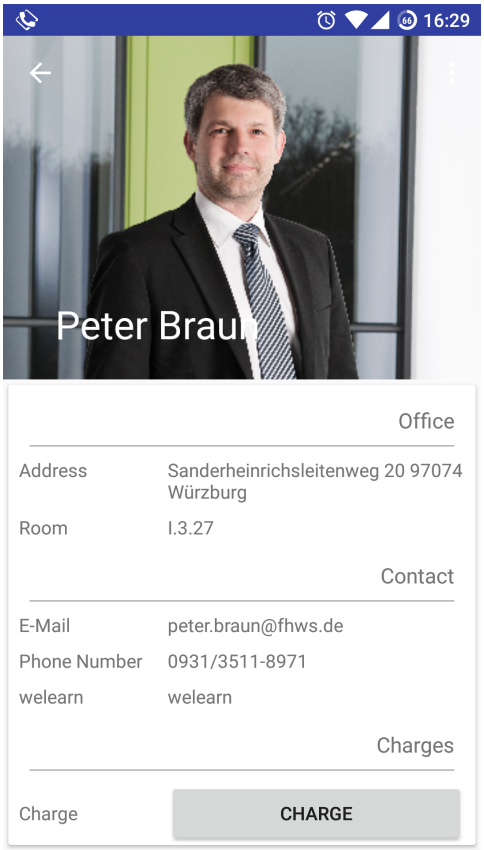

Abbildung 4.5: Beispiel einer DetailView eines Dozenten nach Material Design Guidelines

Die zur Verfügung stehenden Daten sind die gleichen, welche unter Listing 4.1 einzusehen sind.

Analog wie bei der CardView stellen Sich auch bei der DetailView, welche Daten alle dargestellt werden sollen. Hier jedoch gibt es zusätzlich zu der horizontalen Gruppierung (Beispiel mit den Vornamen und Nachnamen), auch noch eine vertikale Gruppierung, die im weiteren auch Kategorisierung genannt wird. In der detaillierten Ansicht eines Dozenten gibt es die Möglichkeit die Attribute zu kategorisieren und jeder Kategorie mit einem Namen zu versehen. Für die Gestaltung und Anordnung sowie mögliche Klick-Aktionen müssen die selben Anforderungen wie bei der CardView berücksichtigt werden.

Jedoch muss die DetailView wissen, welches Attribut den Titel der View darstellt, da dieser in der AppBar erscheinen wird. In diesem Beispiel ist es der Name des Dozenten. Anders als bei der CardView gibt es hier nicht die Möglichkeit zu bestimmen wo das Bild dargestellt werden soll. Ist ein Bild vorhanden, so wird dieses in der Collapsing-Toolbar dargestellt [6]. Andernfalls wird kein Bild angezeigt.

InputView

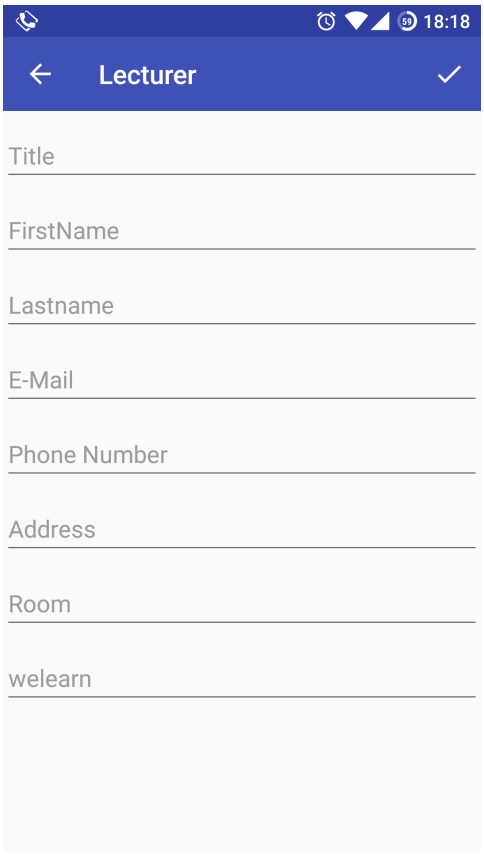

Abbildung 4.6: Beispiel einer View zum Anlegen eines Dozenten

Für das neu Anlegen eines Dozenten oder auch zum bearbeiten muss entschieden werden, welche Attribute zum Anlegen benötigt werden. Auch hier ist es notwendig die Reihenfolge zu bestimmen. Jedoch kommen in dieser View für jedes Attribut noch die Möglichkeit hinzu ein Hint-Text anzugeben. Dieser Text beschreibt, was in der Android View EditText als Beschreibung für das bestimmte Attribut steht. Weiter sollte es die Möglichkeit geben, jedem Feld eine Nachricht mitzugeben, welche Angezeigt wird, wenn das Feld beispielsweise leer gelassen wird. Oder eine weitere Nachricht, wenn das Eingegebene nicht dem Erwarteten entspricht. Zum Beispiel wurde in das Feld für die E-Mail die Telefonnummer eingegeben. Oder es wurde ein regulärer Ausdruck mitgegeben und das Eingegebene entspricht nicht den Anforderungen, welche durch den regulären Ausdruck definiert wurden.

Programmablauf und Klick-Aktionen

Da das Enfield-Model bereits einen endlichen Automaten beschreibt, welcher den Programmablauf widerspiegelt, ist es nicht notwendig, diesen Ablauf noch einmal genauer zu definieren. Da der bereits definierte Ablauf übernommen wird.

Auch die Klick-Aktionen, das Geschehen, welches durch einen Klick auf ein bestimmtes Attribut ausgeführt werden soll, beschränkt sich auf Android Standard Aktionen. Beispielsweise das wechseln zu den Maps, zu einem E-Mail Client, dem Browser oder zum Anrufsmenü. Jede deser Aktion ergibt sich aus den Typen der Attribute, weswegen diese auch nicht weiter definiert werden müssen.

4.1.5 Design der View-Meta-Modelle

In den letzten Abschnitten der Arbeit wurde aufgezählt, was das Meta-Model alles Abdecken muss und das sowohl Android- als auch Polymer-seitig. In diesem Kapitel wird ein Meta-Model vorgestellt, welches die erwähnten Eigenschaften abdeckt.

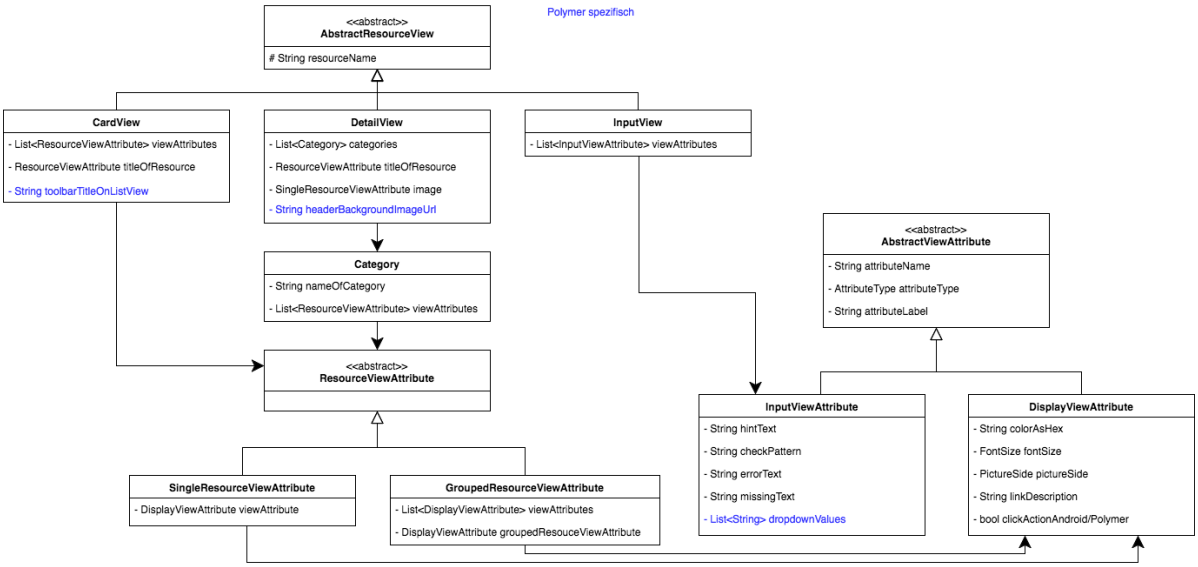

Abbildung 4.7: Aufbau der Views zur Erweiterung des Enfield-Models

Die Abbildung 4.7 zeigt den Aufbau der Objekte, mit welchem das Enfield-Model erweitert wird. Die drei Views CardView, DetailView und InputView sind alles Instanzen von AbstraktResourceView. Jede der View, weiß welche Ressource sie darstellen soll. Dies passiert über die Zuordnung mit Hilfe des Ressourcennamens. Die drei Views, lassen sich

in zwei Kategorien einteilen: Views, welche Informationen anzeigen und Views welche zur Eingabe von Informationen benötigt werden. So gehören CardView und DetailView zur anzeigenden Views und die InputView zur zweiten Kategorie.

Anzeigende Views

Diese View-Typen haben die Aufgabe in einer Liste alle Attribute zu halten, welche in der entsprechenden View angezeigt werden sollen. Dabei bestimmt die Reihenfolge, in welcher die Attribute in dieser Liste sind auch die Anordnung in der Oberfläche. Ist das erste Item in der Liste der Name, so wird dieser ganz oben in der View angezeigt. Bei der DetailView jedoch gibt es nicht eine Liste mit den Attributen, sondern eine Liste mit Kategorien. Diese Kategorien, besitzen einen Namen und eine Liste mit den Attributen ihrer Kategorie. Die Darstellungsreihenfolge der Kategorien und deren Attribute ist analog zu der der CardView. Weiter besitzt die DetailView das Attribute *image*, dieses Attribut wird hier aus der Liste der Attribute herausgezogen, da dieses Attribut bestimmt, ob die View eine Collapsing-Toolbar besitzen wird oder nicht. Wiederum haben beide Views das Attribut *titleOfResource* dieses bestimmt welches Attribut unserer Ressource beispielsweise in der Toolbar angezeigt wird.

Auf die Polymer-spezifischen Attribute wird in dieser Arbeit nicht weiter eingegangen.

Mit Hilfe der Listen, Titelattributen und dem Bildattribut kann das Erscheinungsbild einer View schon ziemlich gut beschrieben werden. Jetzt bleibt fehlt noch die Möglichkeit, die Schriftgrößen, Schriftfarben, Klick-Aktionen und so weiter zu definieren. Außerdem ist es bis jetzt nur möglich einfache Attribute anzuzeigen, eine horizontale Gruppierung ist noch nicht möglich. Um diese Anforderungen zu erfüllen, werden nicht Attribute in den Listen gespeichert sondern Ausprägungen von ResourceViewAttributen.

Es gibt zwei Ausprägungen: ein SingleResourceViewAttribute und ein GroupedResourceViewAttribute. Das SingleResourceViewAttribute ist für einfache Attribute, mit diesem ist es beispielsweise möglich den Titel eines Dozenten anzuzeigen. Das GroupedresourceViewAttribute ermöglicht die horizontale Gruppierung. Beide Objekte, bestimmen jedoch nicht die Design-spezifischen Eigenschaften des Attributs. Hierfür besitzen beide Attribut-Typen das Attribut DisplayViewAttribute.

Bei der SingleResourceViewAttribute ist diese Instanz von einem AbstractViewAttribute das einzige Attribut, beim GroupedResourceViewAttribute wiederum gibt es eine Liste von diesen DisplayViewAttributen, welche dann die anzuzeigenden Informationen widerspiegeln. Weitergehend besitzt diese Attribut-Art auch noch ein DisplayViewAttribute, welches die neu entstandene Gruppierung beschreiben soll.

Ein DisplayViewAttribute besitzt nun die Möglichkeit, die Schriftgröße und -farbe zu

definieren. Die angegebene Farbe muss eine Farbe in hexadezimaler Darstellung sein, wird keine Farbe mitgegeben, wird die Default-Farbe der Anwendung genommen. In der Regel ist diese Schwarz. Die Schriftgröße wiederum ist auf 3 Stufen beschränkt. Es gibt die Möglichkeit den Text in klein, normal und groß darzustellen. Per default ist normal eingestellt. Aus der Oberklasse AbstractViewAttribute besitzt das DisplayViewAttribute noch die Attribute *attributeName*, dieses muss exakt so heißen wie in der Definition der Ressource beschrieben. Mit dem *attributeLabel* kann angegeben werden, wie dieses Attribut in der View angezeigt werden soll. Die Abbildung 4.5 zeigt die Verwendung von den Labels, vor beispielsweise der E-Mailadresse des Dozenten steht *E-Mail*, dieser String entspricht dem Label des Attributes. Muss angeben werden von welchem Typ das aktuell beschriebene Attribut ist. Dies geschickt mit dem Attribut AttributeType. Es gibt folgende mögliche Typen: HOME, MAIL, LOCATION, PICTURE, PHONE_NUMBER, TEXT, URL, DATE, SUBRESOURCE. Jeder Typ bestimmt die Eigenschaften des Attributes. Über diesen wird bestimmt welches Icon in der Karte vor dem entsprechenden Attribut angezeigt werden. Auch bestimmt er welche Aktion bei Klick ausgeführt werden soll. So wird bei einem Klick auf ein Attribut vom Typ LOCATION versucht die Anwendung Maps zu öffnen und den angezeigten Standort dort anzuzeigen. Ist das Attribute vom Typ SUBRESOURCE so wird für dieses Attribut ein Button angezeigt, dieser ermöglicht es dann zu der entsprechenden Subressource zu wechseln. Diese Klick-Aktionen müssen jedoch mit dem Attribut *clickActionAndroid* erst aktiviert werden.

Manche Typen bringen noch ein paar andere Besonderheiten mit sich. So muss man beispielsweise bei einem URL-Attribut noch eine Beschreibung mitgeben, welche anstelle der Hyperlinks angezeigt werden soll. Bei einem Bild kann man beispielsweise noch bestimmen, ob dieses links oder rechts dargestellt werden soll.

Das im Anhang befindliche Listing 1 zeigt die Definition einer DetailView.

Eingebende Views

Bei der InputView gibt es wieder eine Liste, welche dieses mal InputViewAttribute mit der Oberklasse AbstractViewAttribute hält. Diese Liste bestimmt analog zu den anzeigenden Views die darzustellende Reihenfolge der Attribute.

Neben dem *attributeName* der wieder exakt dem Namen aus der Ressourcendefiniton entsprechen muss, besitzt das InputViewAttribute auch die Möglichkeit zu bestimmen, welcher Typ das aktuelle Attribut besitzt. Jedoch haben die Typen hier eine Andere Bedeutung als bei dem anderen View-Typ. So wird beispielsweise bei dem Type DATE kein EditText angezeigt, sondern der Nutzer hat die Möglichkeit das Datum über das DatePicker-Widget von Android einzugeben.

Es ist jedoch für den Android-Client nicht möglich Bilder zu Ressourcen hinzuzufü-

gen, oder diese zu Bearbeiten. Wird eine Subressource nicht in einer InputView der Oberressource bearbeitet oder neu angelegt. Dies ist dann in der entsprechenden View der Subressource möglich. Die anderen Typen beschränken das EditText-Widget auf die angegebenen Typen. So wird bei einem Klick auf ein PHONE_NUMBER-Feld die Tastatur im Zahlenmodus ausgefahren und so weiter.

Einem InputViewAttribute muss zusätzlich ein *hintText* mitgegeben werden, der im EditText des Attributs beschreibt, was in diesem Feld erwartet wird. Mit dem String *missingText* kann dem Attribut mitgegeben werden, welche Nachricht dem Nutzer angezeigt wird, falls er versucht zu speichern ohne das entsprechende Feld auszufüllen. Mit der Kombination von *checkPattern* und *errorText* bekommt der Nutzer des Generators die Validierung des eingegebenen Attributes noch weiter zu verfeinern und auch dem Nutzer der Applikation ein Feedback zu geben, falls eine falsche Eingabe getätigt wurde.

Die Definition einer InputView wird im Anhang unter Listing 2 dargestellt.

4.1.6 Analyse und Design von allgemeinen Daten für eine Anwendung

Dieses Kapitel behandelt die Informationen, welche eine Applikation neben den View-Beschreibungen zusätzlich benötigt, aber diese vom Kontext her nicht in einer der Views beschrieben werden können.

Ein Beispiel für eine solche Information wäre der Uniform Resource Locator (URL) für den Einstieg. Die Applikation benötigt diesen um zu wissen, unter welcher Adresse sich die anzuzeigenden Informationen zu finden sind. Ein weiteres Beispiel sind die Grundfarben der Applikation. Das Material Design gibt drei benötigte Grundfarben vor: *colorPrimary*, *colorprimaryDark* und *colorAccent* diese Grundfarben wird um die Farbe für den Toolbar-Text erweitert.

Mit dem Wissen, konnte eine Erweiterung des Enfield-Models designed werden, welches in Abbildung 4.8 dargestellt ist.

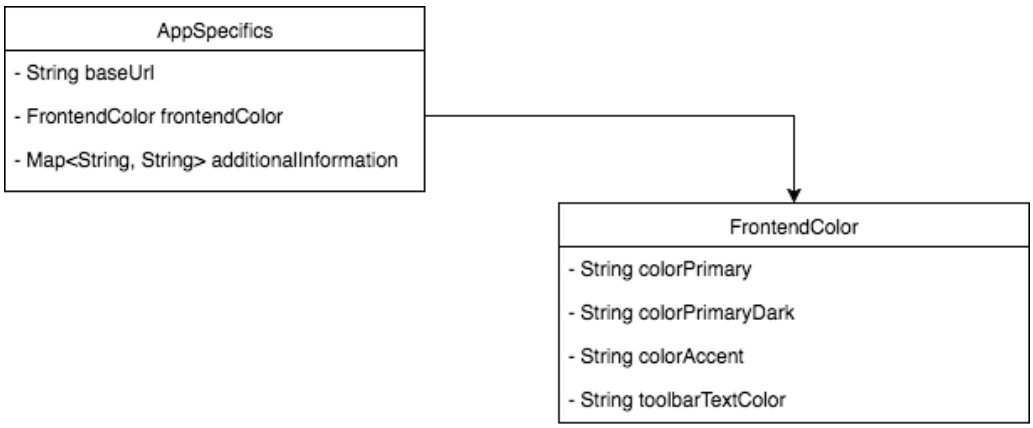

Abbildung 4.8: Aufbau des AppSpecifics Objekt zur Erweiterung des Enfield-Models.

Über die Map *additionalInformation* können zusätzlich weitere allgemeine Informationen an den Generator, zur Erzeugung der Anwendung, weitergegeben werden .

4.2 Software-Generator

Nachdem das Meta-Model nun klar ist, geht dieses Kapitel der Ausarbeitung auf die Funktionsweise des Generators ein. Es wird dargelegt wie der Generator aufgebaut ist und teilweise darauf eingegangen wieso dieser Weg der Generation gewählt wurde. Des weiteren wird das Java Application Programming Interface (API) JavaPoet kurz vorgestellt [10].

4.2.1 JavaPoet

JavaPoet ist ein Java API, welches ermöglicht Java-Klassen zu generieren [10]. Hierfür wird die zu generierende Klasse programmiert. Mit Hilfe von nur ein paar Schlüsselwörtern ist es recht einfach möglich Klassen, Interfaces oder Methoden zu generieren.

Da der größte Teil des Generators Java-Klassen erzeugen muss, ist dieses API bestens für diesen Zweck geeignet. Sie erspart die aufwändige String-Manipulation. Durch die Nutzung wird auch bei der Ausführung des Programmes sichergestellt, das gültige Konventionen und Regeln eingehalten werden. So ist der grundsätzliche korrekte Aufbau einer Java-Klasse bereits sichergestellt.

Listing 4.2 zeigt ein einfaches Beispiel zur Generierung einer Hello-World-Klasse und Listing 4.3 zeigt das Ergebnis nach der Ausführung des Beispieles.

Listing 4.2: Beispiel für die Generation einer Hallo-World-Klasse [10].

```
MethodSpec main = MethodSpec.methodBuilder("main")
    .addModifiers(Modifier.PUBLIC, Modifier.STATIC)
    .returns(void.class)
    .addParameter(String[].class, "args")
    .addStatement("$T.out.println($S)", System.class, "Hello,
   JavaPoet!")
    .build();

TypeSpec helloWorld = TypeSpec.classBuilder("HelloWorld")
    .addModifiers(Modifier.PUBLIC, Modifier.FINAL)
    .addMethod(main)
    .build();

JavaFile javaFile = JavaFile.builder("com.example.helloworld",
   helloWorld)
    .build();
```

Listing 4.3: Ergebnis der Generation von Listing 4.2 [10].

```
package com.example.helloworld;

public final class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, JavaPoet!");
    }
}
```

4.2.2 Generierung anderer Daten-Typen

Neben Java-Klassen besitzt der Sourcecode einer Android Applikation auch XML-Dateien und Gradle-Dateien. Für diese Typen muss eine andere Möglichkeit der Generierung gewählt werden. Hierfür liefert GEnerierung von Mobilen Applikationen basierend auf REST Architekturen (GeMARA) mit der Klasse GeneratedFile eine Möglichkeit. Diese Klasse liefert die die beiden Methoden *append(String contet)* und *appendln(String content)*. Welche es ermöglichen jedes beliebige textbasiertes File-Format zu generieren. Ein GeneratedFile Objekt erzeugt eine Datei, welcher mit den beiden erwähnten Methoden Strings hinzugefügt werden können, dies ermöglicht es jede beliebige Textstruktur zu erzeugen. Jedoch liefert diese Klasse keinerlei Validierung, die Datei wird generiert egal ob die Struktur gültig ist oder nicht.

So würde Listing 4.4 eine Datei *test.xml* im Verzeichnis *generated* erzeugen. Diese erzeuge Datei wird in Listing 4.5 dargestellt.

Listing 4.4: Beispiel eine GeneratedFile-Instanz zur Erzeugung einer XML-Datei.

```
1  public class FileGenerator extends GeneratedFile {
2
3      @Override
4      public void generate() {
5          appendln("<?xml version=\"1.0\" encoding=\"utf-8\"?>");
6          appendln("<menu
       xmlns:android=\"http://schemas.android.com/apk/res/android\"
       xmlns:app=\"http://schemas.android.com/apk/res-auto\">");
7          appendln("<item android:id=\"@+id/saveItem\"");
8          appendln("android:title=\"@string/save\"");
9          appendln("app:showAsAction=\"always\"\\>");
10         appendln("<\\menu>");
11     }
12
13     @Override
14     protected String getFileName() {
15         return "test.xml";
16     }
17
18     @Override
19     protected String getDirectoryName() {
20         return "/generated";
21     }
22 }
```

Listing 4.5: Erzeugte XML-Datei durch den Quellcode von Listing 4.4.

```
1  <?xml version="1.0" encoding="utf-8"?>
2      <menu xmlns:android="http://schemas.android.com/apk/res/android"
3          xmlns:app="http://schemas.android.com/apk/res-auto">
4          <item android:id="@+id/saveItem"
5              android:title="@string/save"
6              android:icon="@drawable/ic_done"
7              app:showAsAction="always"/>
8      </menu>
```

4.2.3 Aufbau der zu generierenden Applikation

Um den Generator möglichst zu vereinfachen, ist es hilfreich, eine Referenzimplementierung der gewünschten Applikation mit all ihren Funktionen und Anforderungen zu entwickeln. Bei einer anschließenden Quellcode-Analyse sollte darauf geachtet werden, die einzelnen Klassen soweit zu abstrahieren, das eine Einteilung in generischen und spezifischen Quellcode erfolgen kann. Der generische Quellcode ist einfacher zu generieren, da dieser statisch ist und sich für alle folgenden Implementierungen nicht verändert. Es können auch Überlegungen angestrebt werden, diese generischen Klassen einfach im Generator abzulegen und bei Bedarf zu kopieren. Diese Methode wurde verworfen, da sich andernfalls jedes mal die kopierten Klassen via String-Manipulation bearbeitet werden müssten. Die minimale Änderung welche jedes mal getroffen werden müsste, wäre das Anpassen der Package Anweisung am Anfang der Java-Klassen und die Anpassung der Import-Anweisungen. Eine weitere Überlegung wäre es, diese Klassen in eine Android Bibliothek auszulagern, und diese dann in jede Anwendung zu importieren. Auch von dieser Möglichkeit wurde in der ersten Version abgesehen, da die Applikation bereits aus zwei Komponenten besteht. Der Applikation an sich und einer Bibliothek, welche die Android-Komponenten für die Anwendung enthält. Um die Komplexität zu reduzieren werden die benötigten generischen Klassen als Teil der eingebunden Bibliothek jedes mal aufs neue generiert.

Die Referenzimplementierung für diese Arbeit beinhaltet folgende Features:

Ressource: Dozent	**Ressource: Amt**
• Anzeige einer Liste mit Dozenten • Anlegen neuer Dozenten • Anzeigen eines Dozenten in Detail • Bearbeiten eines Dozenten • Löschen eines Dozenten	• Anzeigen einer Liste von Ämtern eines Dozenten • Anlegen neuer Ämter für einen Dozent • Anzeigen eines Amtes eines Dozenten • Bearbeiten eines Amtes eines Dozenten • Löschen eines Amtes eines Dozenten

Der Aufbau der Referenzimplementierung wird in Abbildung 1 dargestellt. Das Schaubild verdeutlicht das Verhältnis von generischen (weiße Kästen) und spezifischen (rote

Kästen) Klassen. Die Anzahl der gleichbleibenden Klassen ist mit etwa 60 Prozent bereits höher als der Anteil an spezifischen Klassen. Je höher der Anteil dieser unveränderlichen Klassen, desto geringer wird die Komplexität des Generators. Da der Aufwand eine spezifische Klasse zu erzeugen mehr Logik benötigt, als eine Klasse, welche immer gleich bleibt.

Daneben zeigt die Abbildung 1 aus dem Anhang, auch noch die Aufteilung der Klassen in Klassen der Applikation (gestrichelte Kästen) und Klassen der Bibliothek (solide Kästen). Die Applikation an sich besteht nur aus ein paar wenigen Fragmenten und Aktivities, welche alle projektspezifisch sind. Der komplette generische Quellcode befindet sich in der Bibliothek. Des weiteren befinden sich dort auch die spezifischen Komponenten, beispielsweise der *LecturerInputView*. Diese Komponente, kann in den Fragmenten zur Bearbeitung oder Neuanlage eines Dozenten dann mit wenigen Zeilen Programmcode verwendet werden.

Diese Art der Aufteilung ermöglicht es das ein Applikation Entwickler sich die Komponente, für das Anzeigen, Bearbeiten, Löschen und der Neuanlage generieren lassen kann. Diese Komponenten jedoch beliebig in seiner eigenen Applikation verwenden kann.

4.2.4 Aufbau des Generators

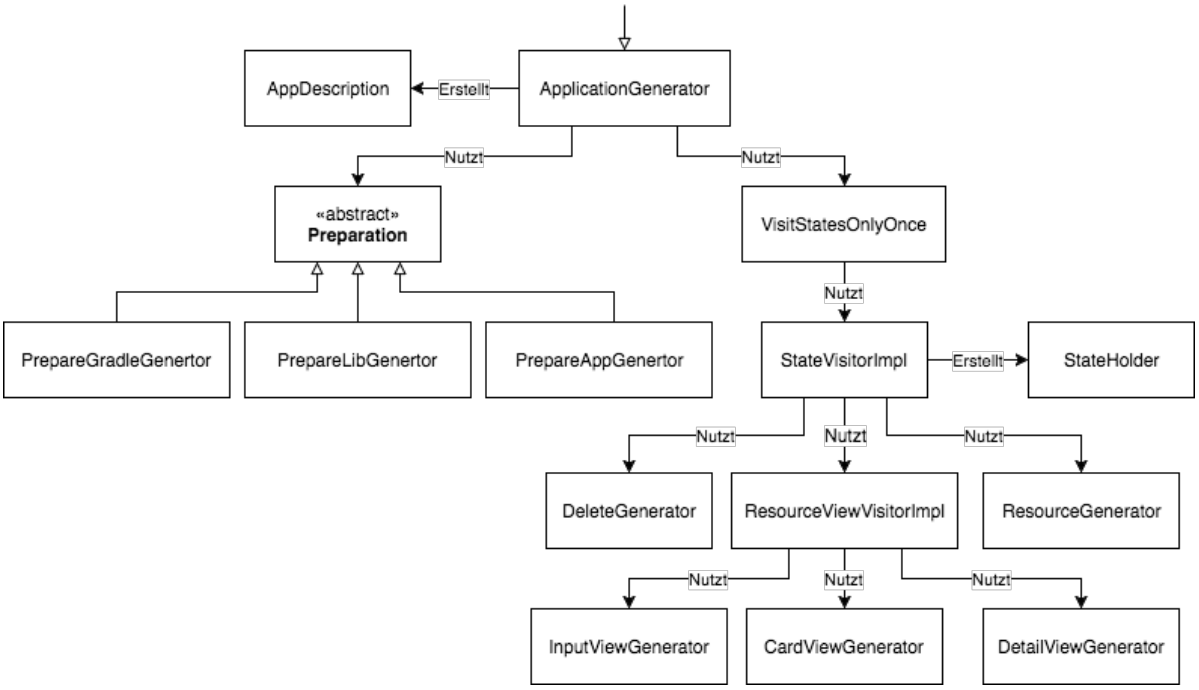

Abbildung 4.9: Aufbau des Android-Generators Welling

Die Klasse ApplicationGenerator, ist der Einstiegspunkt des Projekts. Sie erwartet im Konstruktor ein Enfield-Model Objekt. Wie der Abbildung 4.9 entnommen werden kann, so lässt sich das Projekt in drei Teilbereiche gliedern. Der erste Bereich erzeugt ein AppDescription Objekt (Abbildung 4.10) der zweite Bereich befasst sich mit allgemeinen Vorbereitungen, die getroffen werden müssen. Der Letzte iteriert über die States, und generiert nach bedarf die benötigten Klassen.

Die ApplicationGenerator Klasse verfügt über eine öffentliche Methode *generate*. Beim Aufrufen dieser Methode, werden die einzelnen Generatoren, für den allgemeinen Bereich angestoßen. Weiterhin wird das iterieren über die States des Enfield-Model begonnen. Zum Schluss wird noch das AppDescription Objekt ausgewertet, und die darin enthaltenen Informationen in Dateien geschrieben und an die entsprechende Stelle im Projekt gespeichert.

Erstellung der AppDescription

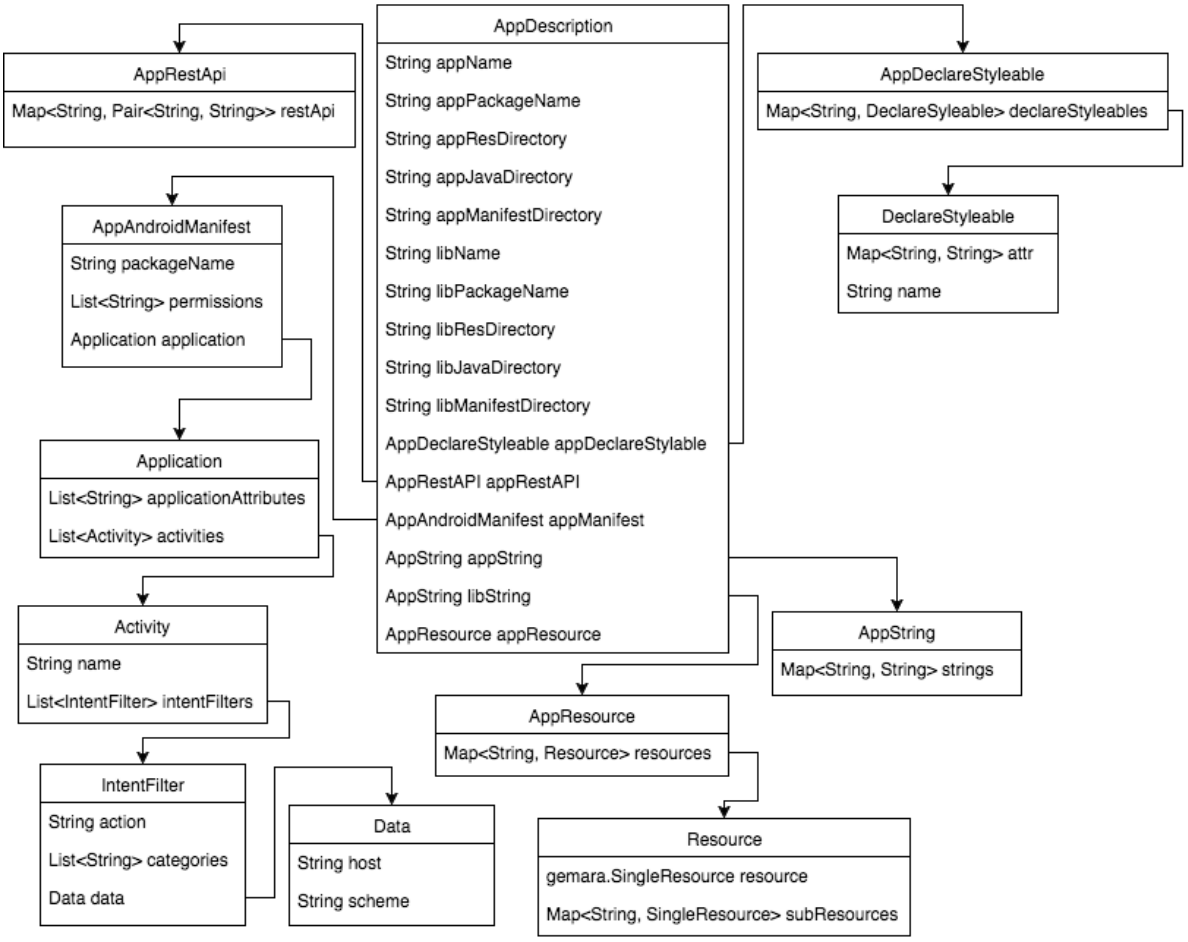

Abbildung 4.10: Aufbau des AppDescription Objekts.

In der AppDescription werden alle Daten durch den Generator gereicht, welche an vielen Stellen benötigt werden. An vielen Stellen wird beispielsweise dir Name der Anwendung oder der Bibliothek benötigt. In jeder Java-Klasse wird der Paket Name benötigt, da dieser in der Bibliothek und in der normalen Applikation verschieden sind müssen diese für beide mitgeführt werden. Auch muss der Generator wissen, unter welchen Verzeichnissen die aktuelle Datei egal ob Java Klasse oder XML-Datei gespeichert werden soll. Diese Dateien können einfach aus dem Enfield-Model abgelesen werden. Auch die Ressourcen und die jeweiligen Subressourcen können direkt aus dem Meta-Model entnommen werden. Dies ist der Teil des initialisieren der AppDescription. Alle bereits jetzt verfügbaren Informationen werden der AppDescription zugewiesen.

Neben diesen Daten, die an mehreren Stellen bei der Generierung benötigt werden, gibt

es Dateien in einer Android-Applikation, die sich mit dem Generieren aufbauen. Ein Beispiel für eine solche Datei ist die *strings.xml*. Es wird in dem generierten Projekt zwei davon geben. Eine im Bereich der Applikation selbst und eine weitere im Bereich der Bibliothek. Diese Dateien enthalten neben dem Applikationsnamen beziehungsweise des Bibliotheksnamen auch viele Strings, die erst beispielsweise in einem Fragment auftauchen. Jedoch müssen die benötigten Datensätze in der *strings.xml* eingetragen werden. Anstelle dies jedes mal wenn im Ablauf des Generierens ein String auftaucht, eine bereits generierte Datei zu erweitern, wird der Datensatz in der AppDescription unter dem AppString *appString* beziehungsweise dem AppString *libString* hinterlegt.

Auch das AndroidManifest wächst mit der Anwendung. So muss jede benutzte Aktivity dort eingetragen sein. Andernfalls kann diese nicht genutzt werden. Am Anfang des Generierens ist die genaue Anzahl und die genauen Namen der Aktivities unbekannt, weswegen der Generator diese beim Erzeugen zur AppDescription hinzufügen muss.

Das Attribut *appDeclareStyleable* enthält alle Custom Attribute, welche wie, im Kapitel 2.2, in die *attr.xml* eingetragen werden müssen.

Da die Anwendung, welche generiert wird auch den REpresentational State Transfer (REST) Ansätzen entsprechen soll, muss diese wissen welche Relationstpyen zu welchen Endpunkten gehören. Anfangs sind diese jedoch auch unbekannt und werden erst im weiteren Verlauf beim iterieren über die States bekannt und zur AppDescription hinzugefügt.

So wächst die AppDescription über den gesamten Prozess des Generierens. Ganz am Ende, werden die gesammelten Daten in die entsprechenden Dateien an den jeweiligen Orten gespeichert. Das Verwenden und weiterreichen eines AppDescription Objekts reduziert die Komplexität des Generators. Dieser muss nicht bei jeder Ergänzung einer der beschriebenen Dateien diese Aufrufen, den neuen Datensatz aufwändig hinzufügen und die Datei wieder abspeichern. Sondern der Generator muss nun die Datei nur einmal schreiben, da er jetzt alle von der Android Anwendung benötigten Informationen besitzt.

Vorbereitung und Generierung allgemeiner Dateien

Der Bereich zur Vorbereitung und Generierung der allgemeinen Dateien gliedert sich ebenfalls in 3 Bereiche. Der erste Bereich kümmert sich um alle Dateien die von Gradle benötigt werden.

Er kopiert Daten wie die *gradlew.bat*, *gradlew*, *build.gradle* und den Gradle Wrapper. Neben dem Kopieren, werden sowohl für die Applikation, Bibliothek als auch für das Gesamtprojekt die spezifischen Dateien generiert. So wird beispielsweise auf der Pro-

jektebene eine *settings.gradle* erzeugt oder in der Applikation sowie in der Bibliothek jeweils eine *build.gradle*.

In der Sektion der Vorbereitung für die Applikation an sich, werden Dateien erzeugt, die jede Applikation benötigt unabhängig von ihrem Aufbau oder den Features. Es wird beispielsweise die MainActivity erzeugt, oder die XML-Dateien, welche für die Transitionsanimationen verantwortlich sind. Auch die *styles.xml* wird erzeugt. Am Schluss werden noch die *mipmap*-Ordner kopiert und an die richtige Stelle verschoben.

Der Bereich, welcher die Bibliothek initialisiert, ist der Größte. Er generiert alle generell benötigten Klassen. Darunter fallen die Klassen für die Netzwerkkommunikation, Die Klasse für das Link-Objekt sowie das Interface *Resource*. Es werden des werden auch die größten Teile der in der Abbildung 1 abgebildeten generischen Klassen erzeugt. Auch werden die grundsätzlichen CustomViews bereits erzeugt. Dazu gehören auch noch die benötigten XML-Dateien. So kann für die Bibliothek beispielsweise das Manifest bereits erzeugt werde, da hier keine Activities registriert werden müssen. Nach dem Ausführen des *PrepareLibGenerators* steht, das Grundgerüst der Bibliothek. Diese enthält nun alle bereits vorab erzeugbaren und benötigten Dateien, welche unabhängig von der gewünschten Funktion der Applikation benötigt werden.

Dieser gesamte Teilbereich des Projekts befasst sich damit ein Grundgerüst für die komplette Android Applikation zu erzeugen und vorab bereits alle benötigten Dateien bereit zu stellen. Die generierten Klassen haben jedoch noch keinerlei Programmlogik, die den spezifischen Ablauf der zu generierenden Anwendung steuert.

Iterieren über die States

Der Teilbereich, der sich mit dem iterieren über die einzelnen States beschäftigt ist der komplexeste Bereich des Generators. Er ist dafür verantwortlich, das zu jedem State die alle benötigten Klassen und Dateien generiert werden.

Um diese Anforderung zu erfüllen, nutzt er den Visitor *IStateVisitor*, welcher durch das Enfield-Model zur Verfügung gestellt wird. Außerdem wird auch der Visitor *VisitStatesOnlyOnce* benutzt. Dieser zweite Visitor stellt sicher, das jeder State nur einmalig Besucht wird. Würde der Generator einfach nur über die Transitionen der States gehen, könnte es passieren, das er in eine Endlosschleife gerät.

Gelangt der Generator zu einem State, wird mit dem *ISateVisitor* identifiziert, von welchem Typ dieser ist. Ist es ein State, welcher einen GET-Request auf eine einzelne Ressource oder auf eine Collection beschreibt, oder beschreibt er einen POST-, PUT- oder DELETE-Request. Nach dieser Identifikation, wird bei jedem State, außer dem DELETE-State, eine Klasse für die in diesem State betroffene Ressource erzeugt. Hierfür

wird der *ResourceGenerator* benutzt. Auch wenn dabei die Ressource mehrfach angelegt werden würde. Der Generator überschreibt eine bereits angelegte Ressource einfach. Diese Redundanz garantiert das auf jeden Fall eine Ressource zum betreffenden State existiert.

Neben diesen Ressource-Klassen, wird auch ein StateHolder-Objekt erstellt. Die Abbildung 4.11 repräsentiert dieses.

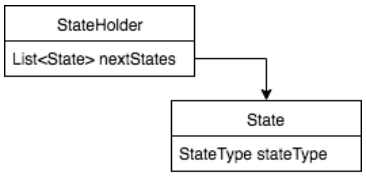

Abbildung 4.11: Aufbau des StateHolder Objekts.

Dieses Objekt wird für jeden einzelnen State angelegt, es enthält alle States, welche über die Transitionen erreicht werden können. So weiß der Generator genau, ob beispielsweise ein Button angezeigt werden muss, der eine Neuanlage einer Ressource ermöglicht. Diese Informationen stecken zwar auch im Enfield-Model, jedoch müsste jedes mal wenn überprüft werden soll welche Folgestates ein State besitzt, über alle States iteriert werden. Das StateHolder-Objekt beschreibt sozusagen eine Landkarte für jeden einzelnen State.

Der State, welcher für das Löschen einer Ressource verantwortlich ist, ist der einfachste zum generieren. Hierfür muss nur ein DialogFragment erzeugt werden, der für das Löschen verwendet wird.

Für die anderen States, werden mehr Klassen und Dateien benötigt. Außerdem werden die ResourceViews (Kapitel 4.1.5) benötigt, die jedem State angehängt sind. Zur Identifizierung der einzelnen ResourceViews wird wiederum mit dem Visitor-Pattern gearbeitet. Die Klasse der ResourceView stellt den Visitor „ResoruceViewVisitor“ zur Verfügung. Nachdem bekannt ist welche der drei ResourceView-Typen im entsprechenden State verwendet wurde, kann einer der Komponentengeneratoren: InputViewGenerator, CardViewGenerator oder DetailViewGenerator alle notwendigen Dateien generieren.

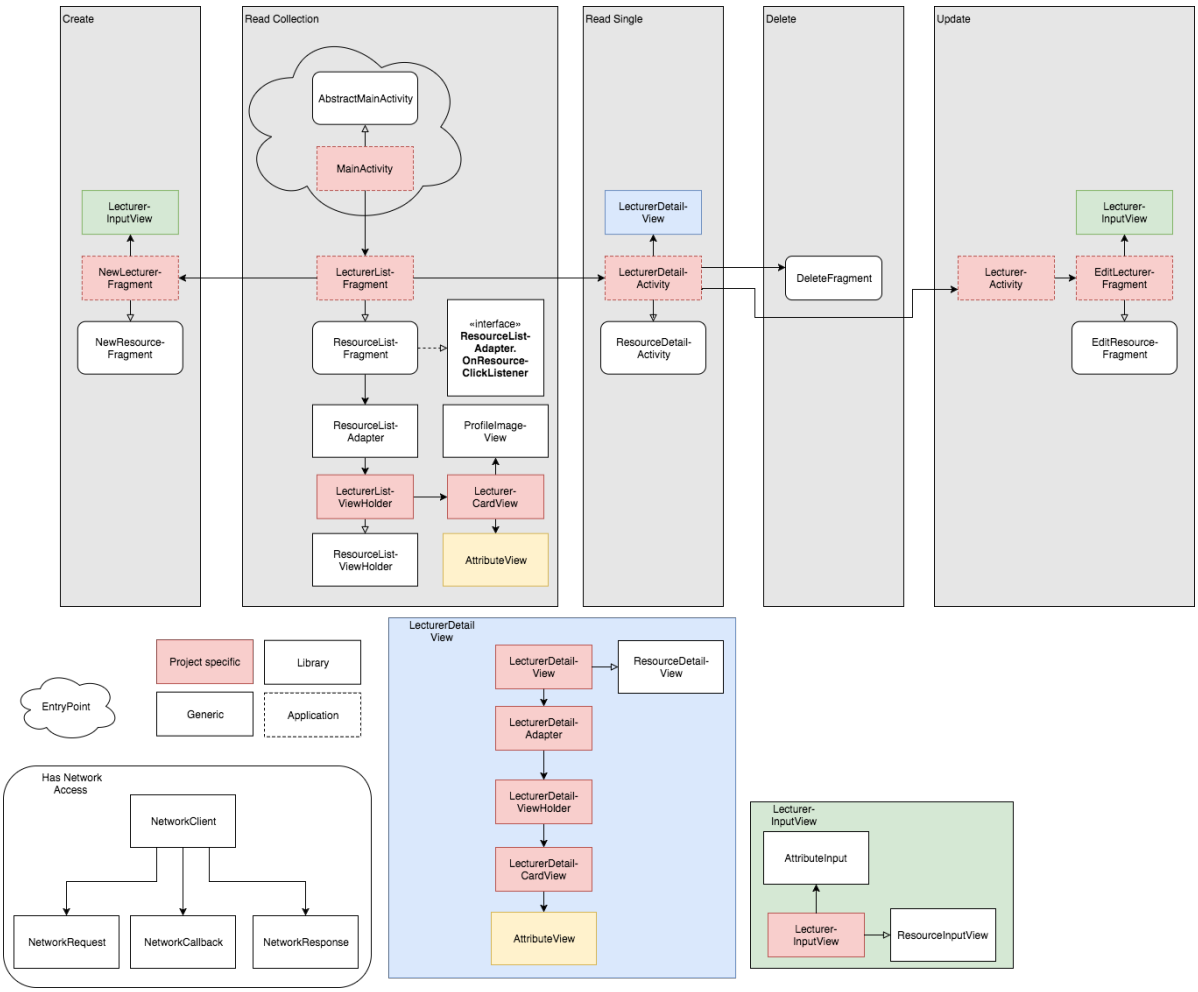

Abbildung 4.12: Aufbau der Dozenten Applikation mit Einteilung in spezifische States.

In Abbildung 4.12 ist die Applikation für Dozenten noch einmal abgebildet. Zur Vereinfachung wurde bei diesem Diagramm jedoch die Ressource Ämter mit ihren zugehörigen Klassen weggelassen.

Der Bereich „Update“ und der Bereich „Create“ werden hierbei vom InputViewGenerator, der Bereich „Read Collection“ von CardViewGenerator und der Bereich „Read Single“ vom DetailViewGenerator erzeugt.

Jeder der einzelnen Generatoren ist ein Zusammenschluss von vielen Teilgeneratoren. Es werden dabei in einem der Generatoren nicht nur die Java-Klassen für die Applikation oder die Bibliothek, sondern auch alle benötigten XML-Dateien erzeugt.

So ist beispielsweise der DetailViewGenerator dafür verantwortlich, dass auf der Seite der Applikation, die „LecturerDetailActivity“ inklusive ihrer XML-Datei erzeugt wird. Er

muss weitergehend auch diese Aktivity in die AppDescription im Bereich des Manifestes hinterlegen. Im Bereich der Bibliothek muss dafür gesorgt werden, dass die generischen Klassen „ResourceDetailActivity“, „ResourceDetailView“ sowie die spezifischen Klassen: „LectuererDetailView“, „LectuererDetailAdapter“, „LectuererDetailViewHolder“, „LectuererDetailCardView“ erzeugt werden. Zu all diesen Klassen müssen mögliche Strings oder CustomViews in die AppDescription aufgenommen werden. Wiederum müssen auch die entsprechenden XML-Dateien erzeugt werden.

Jeder Generator besitzt mehrere Möglichkeiten, welche Klassen generiert werden müssen. So entscheidet beispielsweise ob die Ressource ein Bild besitzt oder nicht über den Verhalt, ob eine Activity mit einer CollapsingToolbar verwendet wird oder ob ein einfaches Fragment zur Detailanzeige ausreichend ist.

Selbst die Generatoren auf der untersten Ebene, welche die einzelne Klassen erzeugen, wissen mit Hilfe von dem mitgegebenen StateHolder, ob beispielsweise Menüeinträge für das Löschen oder das Bearbeiten von Ressourcen benötigt werden. Diese Generatoren richten sich auch nach den übergebenen RessourceViews. Auf dieser Ebene haben die vom Benutzer des Generators mitgegebenen Informationen zum Aussehen, Einfluss. Hier werden die benötigten Attribute der Ressource hinzugefügt, und deren Aussehen in den entsprechenden XML-Dateien beschrieben.

4.3 Bauen und ausführen der generierten Android Applikation

Wurden alle benötigten Dateien der Applikation erzeugt, gibt es zwei Möglichkeiten, die Applikation zu bauen und anschließend auf einem Android-Endgerät zu installieren.

Variante 1: Importieren der generierten Dateien in eine **ide!** (**ide!**) beispielsweise in Android Studio. Dort wie bereits bekannt, die Anwendung bauen und auf einem sich im Entwicklermodus befindlichen Android-Endgerät installieren.

Variante 2: Die Applikation mit Hilfe des Makefile bauen und installieren. Hierfür muss ebenfalls ein Android-Endgerät im Entwicklermodus an dem entsprechenden Computer angeschlossen sein.

Listing 4.6: Makefile für das Bauen und Installieren der erzeugten Applikation.

```
1 APK =
    gemara/android/src-gen/generated/app/build/outputs/apk/app-debug.apk
2
3 all: debug install
4
5 debug:
6 cd gemara/android/src-gen/generated && chmod 777 gradlew && ./gradlew
    clean assembleDebug
7
8 install:
9 adb $(TARGET) install -rk $(APK)
```

Listing 4.6 zeigt das Makefile, dieses bietet die Möglichkeit entweder mit dem Befehl *make* eine Debug-Version der Anwendung zu bauen und zu Installieren, oder mit dem Befehl *make debug* ausschließlich die Applikation zu bauen beziehungsweise mit dem Befehl *make install* die bereits gebaute Applikation zu installieren.

5 Evaluierung

//Todo

6 Zusammenfassung

//Todo

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] John Abou-Jaoudeh u. a. „A High-Level Modeling Language for the Efficient Design, Implementation, and Testing of Android Applications“. In: *arXiv preprint arXiv:1508.02153* (2015).

[2] *Androiden dominieren den Smartphone-Markt*. Eingesehen am 12.11.16. URL: https://de.statista.com/infografik/902/weltweiter-marktanteil-der-smartphone-betriebssysteme/.

[3] *AppBrain. Anzahl der verfügbaren Apps im Google Play Store in ausgewählten Monaten von Dezember 2009 bis Oktober 2016 (in 1.000)*. Eingesehen am 12.11.16. URL: https://de.statista.com/statistik/daten/studie/74368/umfrage/anzahl-der-verfuegbaren-apps-im-google-play-store/.

[4] *Architectural Styles and the Design of Network-based Software Architectures, Chapter 5*. eingesehen am 17.11.16. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[5] *ARD/ZDF-Onlinestudie 2016: 84 Prozent der Deutschen sind online ? mobile Geräte sowie Audios und Videos mit steigender Nutzung*. Eingesehen am 12.11.16. URL: http://www.ard-zdf-onlinestudie.de/.

[6] *CollapsingToolbarLayout*. eingesehen am 28.02.17. URL: https://developer.android.com/reference/android/support/design/widget/CollapsingToolbarLayout.html.

[7] *Crud Admin Generator*. eingesehen am 17.11.16. URL: http://crud-admin-generator.com/.

[8] *DomainSpecificLanguage*. eingesehen am 24.02.17. URL: https://martinfowler.com/bliki/DomainSpecificLanguage.html.

[9] Paul Hudak. „Domain-specific languages“. In: *Handbook of Programming Languages* 3.39-60 (1997), S. 21.

[10] *JavaPoet*. eingesehen am 02.03.17. URL: https://github.com/square/javapoet.

[11] *Material Design*. eingesehen am 28.02.17. URL: https://material.io/guidelines/.

[12] *open handset alliance*. eingesehen am 06.01.17. URL: http://www.openhandsetalliance.com/index.html.

[13] *TechCrunch. (n.d.). Anzahl der im Apple App Store verfügbaren Apps von Juli 2008 bis Juni 2016. In Statista - Das Statistik-Portal.* Eingesehen am 12.11.16. URL: `https://de.statista.com/statistik/daten/studie/20150/umfrage/anzahl-der-im-app-store-verfuegbaren-applikationen-fuer-das-apple-iphone/`.

[14] *Website (internetdo.com). Prognose zur Anzahl der Smartphone-Nutzer weltweit von 2012 bis 2020 (in Milliarden).* Eingesehen am 12.11.16. URL: `https://de.statista.com/statistik/daten/studie/309656/umfrage/prognose-zur-anzahl-der-smartphone-nutzer-weltweit/`.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorgelegte Bachelorarbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Marcel Groß, am 4. März 2017

Anhang

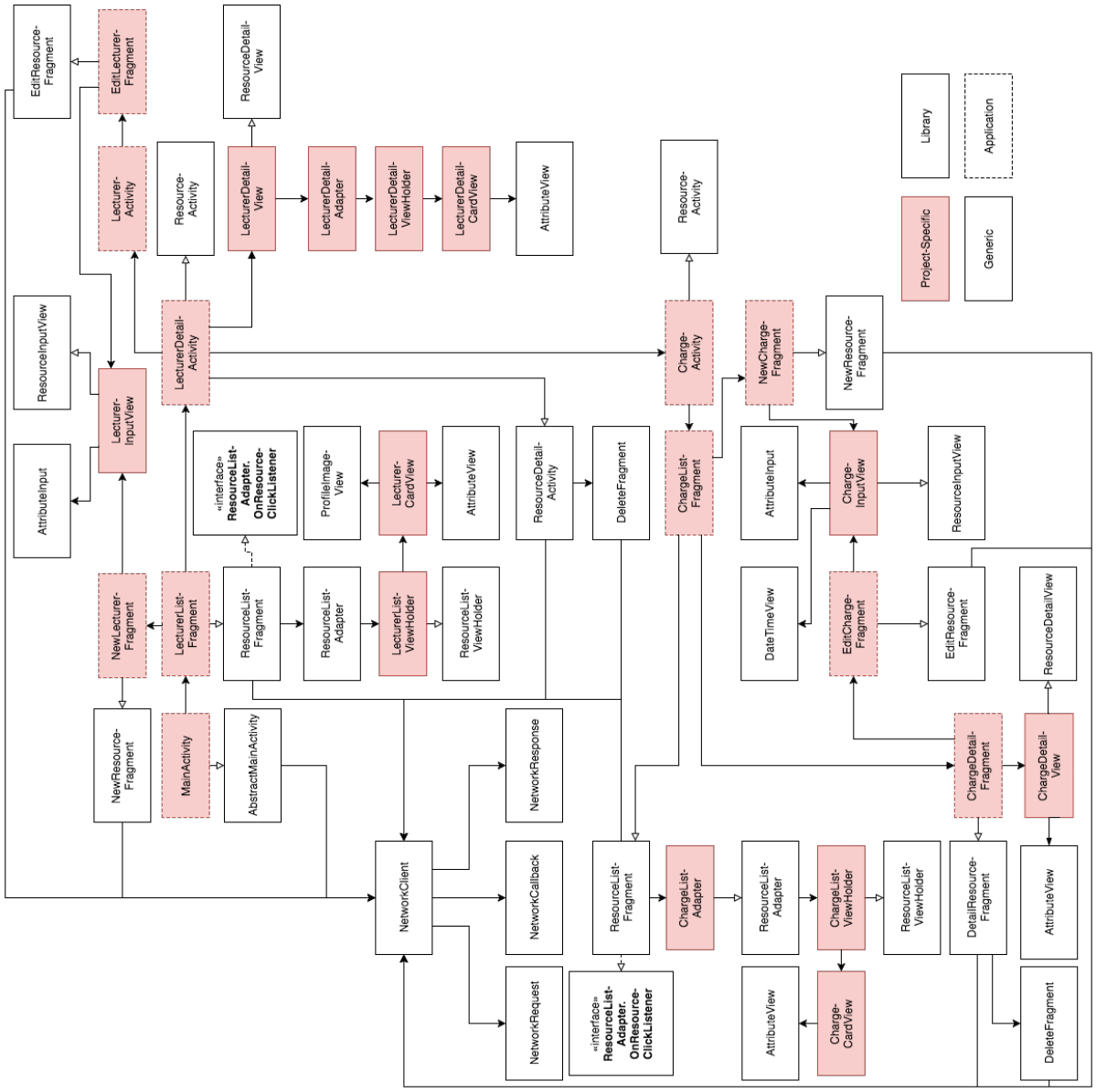

Abbildung 1: Aufbau der Referenzimplementierung

Listing 1: Erstellung einer DetailView.

```
...
DetailView detailView;

try {
List<Category> categories = new ArrayList<>();

DisplayViewAttribute nameAttribute = new DisplayViewAttribute("name",
    ViewAttribute.AttributeType.TEXT);
GroupResourceViewAttribute name = new
    GroupResourceViewAttribute(nameAttribute, getViewTitleAttributes());

categories.add(new Category("Office",
    getOfficeResourceViewAttributes()));
categories.add(new Category("Contact",
    getContactResourceViewAttributes()));
categories.add(new Category("Charges",
    getChangeResourceViewAttributes()));

detailView = new DetailView("Lecturer", name, categories);
detailView.setImage(getImage());
} catch (DisplayViewException ex) {
detailView = null;
}
...
private static List<ResourceViewAttribute>
    getOfficeResourceViewAttributes() {
List<ResourceViewAttribute> officeAttributes = new ArrayList<>();

DisplayViewAttribute addressAttribute = new
    DisplayViewAttribute("address",
    ViewAttribute.AttributeType.LOCATION);
addressAttribute.setAttributeLabel("Address");
addressAttribute.setClickActionAndroid(true);
SingleResourceViewAttribute address = new
    SingleResourceViewAttribute(addressAttribute);
officeAttributes.add(address);

DisplayViewAttribute roomAttribute = new
    DisplayViewAttribute("roomNumber", ViewAttribute.AttributeType.TEXT);
roomAttribute.setAttributeLabel("Room");
roomAttribute.setClickActionAndroid(true);
SingleResourceViewAttribute room = new
```

```
    SingleResourceViewAttribute(roomAttribute);
officeAttributes.add(room);

return officeAttributes;
}
...
```

Listing 2: Erstellung einer InputView.

```
1  ...
2  List<InputViewAttribute> inputViewAttributes = new ArrayList<>();
3
4  InputViewAttribute title = new InputViewAttribute("title",
       ViewAttribute.AttributeType.TEXT, "Title", "Title is missing!");
5  title.setAttributeLabel("Title");
6  inputViewAttributes.add(title);
7
8  InputViewAttribute firstName = new InputViewAttribute("firstName",
       ViewAttribute.AttributeType.TEXT, "FirstName",
9  "Firstname is missing!");
10 firstName.setAttributeLabel("Firstname");
11 inputViewAttributes.add(firstName);
12
13 InputViewAttribute lastName = new InputViewAttribute("lastName",
       ViewAttribute.AttributeType.TEXT, "Lastname",
14 "LastName is missing!");
15 lastName.setAttributeLabel("Lastname");
16 inputViewAttributes.add(lastName);
17
18 InputViewAttribute mail = new InputViewAttribute("email",
       ViewAttribute.AttributeType.MAIL, "E-Mail", "E-Mail is missing!");
19 mail.setAttributeLabel("E-Mail");
20 inputViewAttributes.add(mail);
21
22 InputViewAttribute phone = new InputViewAttribute("phone",
       ViewAttribute.AttributeType.PHONE_NUMBER, "Phone Number",
23 "Phone number is missing!");
24 phone.setAttributeLabel("Phone Number");
25 inputViewAttributes.add(phone);
26
27 InputViewAttribute address = new InputViewAttribute("address",
       ViewAttribute.AttributeType.TEXT, "Address", "Address is missing!");
28 address.setAttributeLabel("Address");
29 inputViewAttributes.add(address);
30
31 InputViewAttribute room = new InputViewAttribute("roomNumber",
       ViewAttribute.AttributeType.TEXT, "Room", "Room is missing!");
32 room.setAttributeLabel("Room");
33 inputViewAttributes.add(room);
34
35 InputViewAttribute weLearn = new InputViewAttribute("homepage",
```

```
    ViewAttribute.AttributeType.URL, "welearn",
"welearn URL is missing!");
weLearn.setAttributeLabel("welearn");
inputViewAttributes.add(weLearn);

InputView inputView;
try {
inputView = new InputView("Lecturer", inputViewAttributes);
} catch (InputViewException ex) {
inputView = null;
}
...
```