

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

Der wachsende Markt für Android Applikationen erfordert immer schnellere Entwicklungs- und Releasezeiten. Viele dieser Anwendungen haben den gleichen Grundaufbau, sie ermöglichen Daten in Listen oder in einer Detailansicht anzuzeigen und gegebenenfalls diese Daten zu bearbeiten. Durch diesen ähnlichen Grundaufbau ist es möglich, diese Applikationen durch Software-Generatoren erzeugen zu lassen. Die Erzeugung von Android Applikationen mit Hilfe eines Software-Generators birgt einige Vorteile. So können durch Zuhilfenahme von Generatoren diese Applikationen mit sehr wenig Quellcode beschrieben und erzeugt werden. Der Schwerpunkt dieser Arbeit liegt darin ein grundsätzliches Verständnis für die Entwicklung von CustomViews in Android und in Design eines Software-Generators zu schaffen.

Abstract

The growing market for Android applications requires faster development and release times. Many of these applications have the same basic structure, they display data in lists or in a detailed view and, if necessary, the user can edit this data. Because of this similar basic structure, it is possible to have these applications generated by software generators. The generation of Android applications using a software generator has some advantages. By using generators, these applications can be described and generated with very little source code. The main focus of this thesis is to create a basic understanding of the development of CustomViews in Android and the design of a software generator.

Danksagung

Ich möchte mich bei meinem betreuenden Professor Dr. Peter Braun bedanken. Er hatte immer ein offenes Ohr für mich und ich konnte mit ihm über auftretende Probleme diskutieren und diese dadurch lösen. Durch seine Erfahrung bei der Entwicklung von GEneration von Mobilen Applikationen basierend auf REST Architekturen (GeMARA) konnte er mir wertvolle Ratschläge und Ansätze für meine Implementierung geben.

Auch möchte ich Markus Fisch danken, der mir bei Problemen mit der Entwicklung der Android Applikation immer zur Seite stand. Dankenswerterweise hat er auch die ein oder andere Stunde mit mir die Applikation debugged, um noch die kleinsten Fehler zu finden. Egal ob es ein Fehler im generellen Aufbau der Anwendung war oder im Programmcode.

Zuletzt möchte ich mich bei allen weiteren Unterstützern bedanken. Ob es Hilfe bei Problemen mit LaTeX war oder beim Rat zum Aufbau der Arbeit. Auch allen Korrekturlesern möchte ich nochmal besonders danken. Ohne euch würde diese Arbeit nicht so flüssig sein.

Inhaltsverzeichnis

1	Einführung	1
1.1	Motivation	3
1.2	Zielsetzung	3
1.3	Aufbau der Arbeit	4
2	Grundlagen	6
2.1	<i>REpresentational State Transfer (REST)</i>	6
2.2	Entwicklung von Android <i>CustomViews</i>	7
2.2.1	Registrieren der <i>CustomView</i>	8
2.2.2	Definieren des Aufbaus der <i>CustomView</i>	9
2.2.3	Erzeugen einer <i>CustomView</i> Klasse	9
2.3	Software-Generatoren	11
2.3.1	<i>Domänenspezifische Sprache (DSL)</i>	11
2.3.2	GEnerierung von Mobilien Applikationen basierend auf REST Ar- chitekturen (GeMARA)	12
3	Problemstellung	14
3.1	Vorstellung der Referenzanwendung	14
3.1.1	Backend Referenzimplementierung	15
3.1.2	Android Referenzimplementierung	18
3.2	Analyse der Android Anwendung	23
3.2.1	Analyse des Aufbaus	23
3.2.2	Analyse der Android <i>Views</i>	25
3.3	<i>Meta-Modell</i>	26
3.3.1	Kompatibilität mit GeMARA und andern möglichen Clients	26
3.3.2	Eigenes <i>Android-Meta-Modell</i>	27
3.3.3	Allgemeine Erweiterungen des <i>Enfield-Modells</i>	28
3.3.4	Analyse der benötigten Dateien für das <i>Meta-Modell</i>	29
3.3.5	Design der <i>View-Meta-Modelle</i>	34
3.3.6	Analyse und Design allgemeiner Daten für eine Anwendung	40
4	Lösung	41
4.1	Design des Software-Generators	41
4.1.1	<i>JavaPoet</i>	41
4.1.2	Generierung anderer Datentypen	42

Inhaltsverzeichnis

4.1.3	Ablauf der Generierung	44
4.1.4	Aufbau des Generators	45
4.2	Bauen und Ausführen der generierten Android Applikation	51
5	Evaluierung anhand einer Beispielanwendung	53
5.1	Erstellung und Nutzung des <i>Meta-Modells</i>	53
5.2	Zeitaufwände und Komplexität	54
6	Zusammenfassung	56
6.1	Zusammenfassung	56
6.2	Ausblick	57
	Verzeichnisse	59
	Literatur	61
	Eidesstattliche Erklärung	62
	Anhang	63

1 Einführung

Das Smartphone ist heutzutage der stete Begleiter eines Menschen. „Zwei Drittel der Bevölkerung und nahezu jeder 14- bis 29-Jährige geht darüber ins Netz.“ [4] Auch die Prognose zeigt, dass der Absatzmarkt immer weiter steigen wird (Abbildung 1.1).

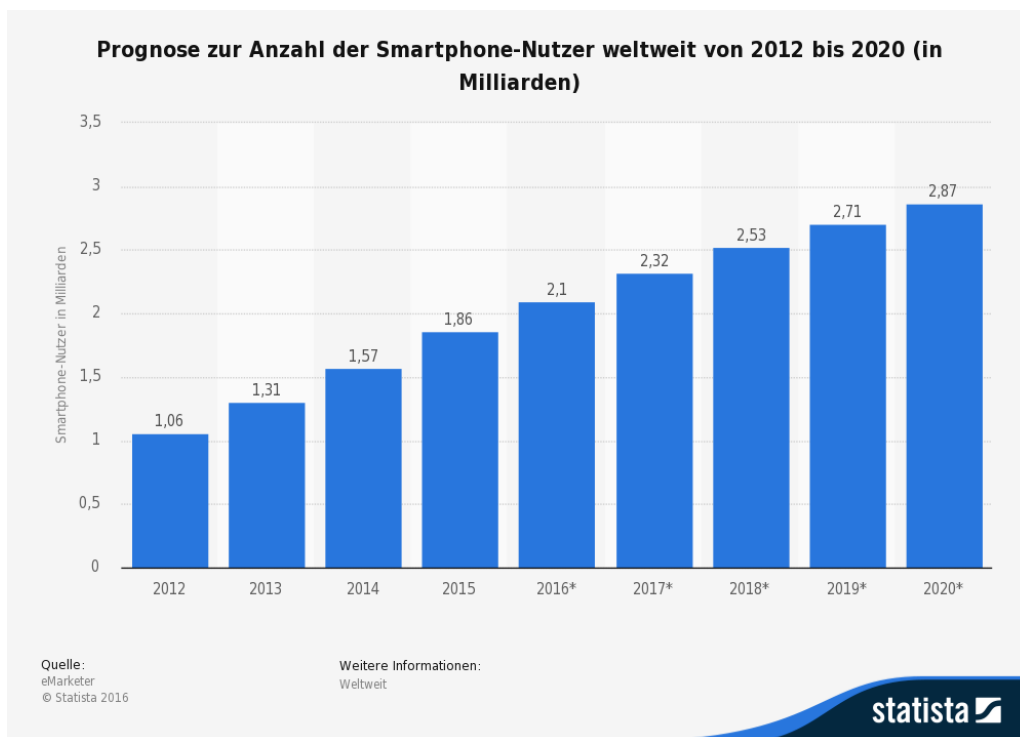


Abbildung 1.1: Prognose zur Anzahl der Smartphone-Nutzer weltweit von 2012 bis 2020 (in Milliarden) [17].

Umso wichtiger ist, dass 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 und ist der Grund weswegen in den einzelnen Stores heutzutage so viele Apps angeboten werden. Bei Android im Playstore sind im Oktober 2016 ca. 2,4 Millionen Apps [3], bei Apple im App Store sind es ca. 2 Millionen Apps (Stand Juni 2016) verfügbar [16]. Neben Googles Android

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

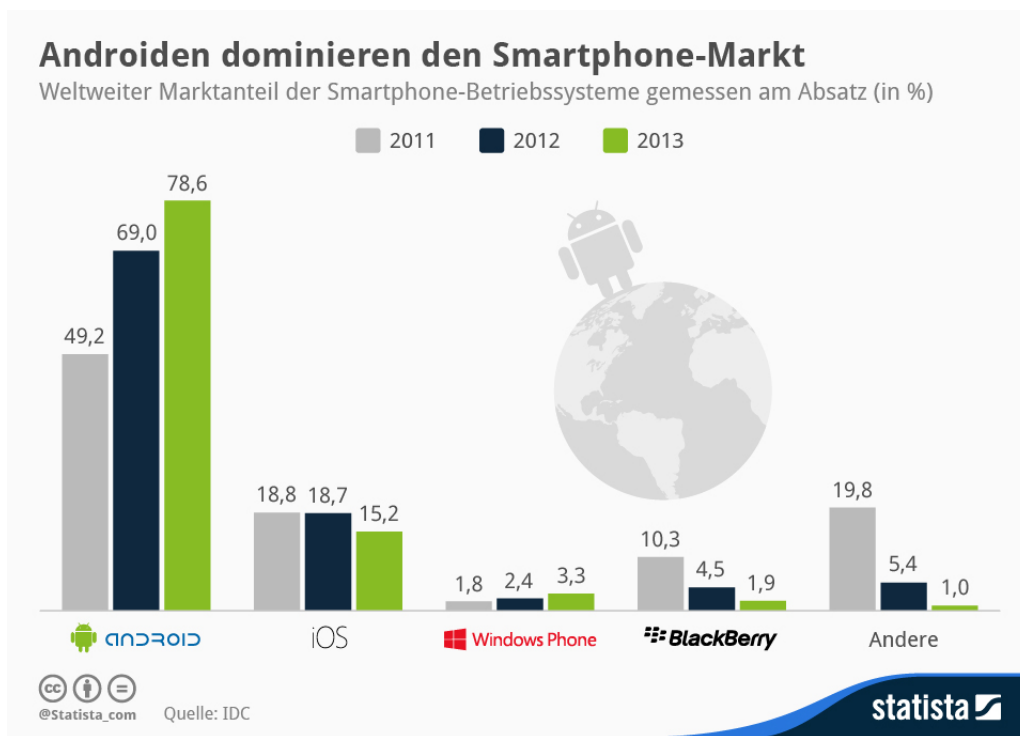


Abbildung 1.2: Der weltweite Marktanteil von Smartphone-Betriebssysteme [2].

Jede dieser Applikationen wurde 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 Codepassagen in jeder dieser Anwendungen vorhanden sind, welche einen ähnlichen beziehungsweise denselben Zweck erfüllen. Werden diese Stellen im Programmcode abstrahiert, gibt es die Möglichkeit diese generieren zu lassen. Um Code generieren zu lassen, werden so genannte Code-Generatoren benötigt.

Im Bereich der Backend-Entwicklung gibt es bereits verschiedene Projekte die sich damit befassen. Ein Beispiel wäre der *CRUD Admin Generator* [6]. 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 verschiedener Anforderungen nicht länger 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 Activities* beschäftigt. Die dabei entstandenen Lösungen, resultieren darin, dass das generieren von *Activities* zu Problemen führt. Deshalb beschäftigt sich diese Ausarbeitung damit, nicht eine komplette *Activity* 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, kann eine Bibliothek mit den erzeugten Komponenten allen Studierenden zur Verfügung gestellt werden. Dadurch wäre jeder Studierende 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 *REpresentational State Transfer (REST)* vertieft werden. Hierbei wird ein Schwerpunkt auf das *Hypermedia-Prinzip* gelegt.

Neben diesen spezifischen Anforderungen soll ein Verständnis 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 das *User Interface (UI)*, welches ebenfalls generiert wird, auch generisch gestaltet werden kann. Das bedeutet, dass nicht nur Informationen, welche angezeigt werden sollen beschrieben werden, sondern auch wie diese angezeigt werden sollen.

Wenn es möglich ist, dass das *UI* als Teil der *domänenspezifischen Sprache (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

Diese Ausarbeitung ist in sieben Kapitel unterteilt. In der Einführung wird zu Beginn auf den Stellenwert von Android Applikationen eingegangen. Darauf folgt die Motivation, weswegen diese Arbeit geschrieben wurde. Diese soll die Problemstellung anreißen und zur Zielsetzung hinführen. In diesem Teil der Einführung wird definiert, was der Sinn dieser Arbeit ist. Das Kapitel wird dann mit dem Bereich abgeschlossen, welchen den Aufbau der Arbeit beschreibt.

Das zweite Kapitel befasst sich mit den Grundlagen. Hier soll der Leser noch einmal seinen Kenntnisstand über *REpresentational State Transfer (REST)* auffrischen und die Bedeutung von *Hypermedia as the Engine of Application State (HATEOAS)* verstehen. Neben dem Bereich der Netzwerkkommunikation wird außerdem noch der Bereich Android angeschnitten. Hier liegt der Schwerpunkt in der Entwicklung und *CustomViews*. Dabei werden die einzelnen Schritte aufgezeigt, die benötigt werden, um diese Komponenten zu erstellen und zu benutzen. Der letzte Teil in den Grundlagen befasst sich mit Software-Generatoren. Hier wird dem Leser kurz erklärt, was eine *domänenspezifische Sprache (DSL)* ist und welche zwei generelle Arten es gibt. Im Anschluss wird das Projekt GEneration von Mobilen Applikationen basierend auf REST Architekturen (GeMARA) vorgestellt.

Im dritten Kapitel wird die Problemstellung behandelt, dabei wird die Referenzanwendung vorgestellt. Diese Vorstellung inkludiert sowohl das Backend sowie die Android Applikation. Es wird darauf eingegangen, dass das Backend mit Hilfe von GeMARA generiert wurde und wie das daraus resultierende Application Programming Interface aussieht. Anschließend wird die Anwendung dahingehend analysiert, dass eine Einteilung in generischen sowie spezifischen Quellcode vorgenommen werden kann. Auch werden die einzelnen *Views* analysiert. Diese Analyse sollen Gemeinsamkeiten beim Aufbau und Design offenbaren, so dass diese mit einer *DSL* beschrieben werden können. Zum Abschluss wird das *Meta-Modell* vorgestellt. Es wird auf die Anforderungen an dieses eingegangen und anschließend zwei Modelle vorgestellt. Ein reines Android Modell und dann die

Erweiterung des vorhandenen Enfield-Modells. Der nächste Bereich befasst sich mit der Analyse, hierbei soll herausgefunden werden, welche Daten das *Meta-Modell* überhaupt benötigt. In der Analyse werden die einzelnen *Views* genauer betrachtet und es wird ein Augenmerk auf den Programmablauf und die Aktionen bei Klick gelegt. Nach der Analyse wird der Aufbau der *View-Meta-Modelle* vorgestellt. Bei jeder *View* wird auf deren Besonderheit und Möglichkeiten eingegangen. Neben den *View-spezifischen* Daten wird auch noch aufgezeigt, welche Dateien allgemein benötigt werden und wo diese im vorgegebenen Modell platziert werden sollten.

Das Kapitel Lösung stellt *Welling* vor. Dies ist der in dieser Arbeit entwickelte Software-Generator. Zunächst wird das *Java Application Programming Interface (API) JavaPoet* im Zusammenhang mit der Generierung von Java-Klassen vorgestellt, anschließend wird beschrieben, wie andere Datei-Typen generiert werden können. Es wird gezeigt welche Features unterstützt werden müssen. Nachfolgend wird der Aufbau des Generators vorgestellt. Es wird auf die einzelnen Bereiche eingegangen und deren Aufgabe sowie Funktionsweise erklärt. Abgeschlossen wird das Kapitel mit einer Anleitung, wie die generierte Applikation gebaut und ausgeführt werden kann.

Das fünfte Kapitel, Evaluierung anhand einer Beispielanwendung, ist in drei Bereiche eingeteilt. Zunächst wird die Beispielanwendung vorgestellt. Anschließend wird auf die Erstellung und Nutzung des *Meta-Modells* eingegangen, wobei hier auch Einschränkungen durch dieses aufgezeigt werden. Der letzte Bereich befasst sich dann noch einmal mit den Zeitaufwänden und der Komplexität der Entwicklung, Wartung und Nutzung des Generators. Auch wird die Komplexität der erzeugten Applikation kritisch bewertet.

Im letzten Kapitel, Zusammenfassung, wird die Arbeit noch einmal reflektiert. Zusätzlich noch mögliche Erweiterungen und Ergänzungen an *Meta-Modell* und Software-Generator vorgestellt.

2 Grundlagen

2.1 *RE*presentational *State* Transfer (*REST*)

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

REST [7] ist ein Programmierparadigma, welches sich auf folgende Prinzipien stützt: *Client-Server*, *Zustandslose Kommunikation*, *Caching*, *Uniform Interface*, *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 [14]. Der Architekt einer *REST*-konformen *Application Programming Interface (API)* überlegt sich im Voraus, wie der Applikations-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 dieses dem *Client-Entwickler* als Anleitung zum Erstellen des *Clients* dienen. Er benötigt diesbezüglich 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 *Transitionen* 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 *Transition* zum *State* „Collection“ zurück. Dieser *State*, verfügt über alle Informationen, die benötigt werden, um eine Collection der betroffenen Ressource anzuzeigen, weiterhin hat 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. Von hier kann die Anwendung zum *State* „Update“ oder „Delete“ wechseln. Der *State* „Update“ ermöglicht 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, dass der Entwickler nur den Einstiegspunkt „Dispatcher“ kennen muss. Die Anwendung liefert selbst alle benötigten Informationen, um die Daten für die Anwendung nachzuladen.

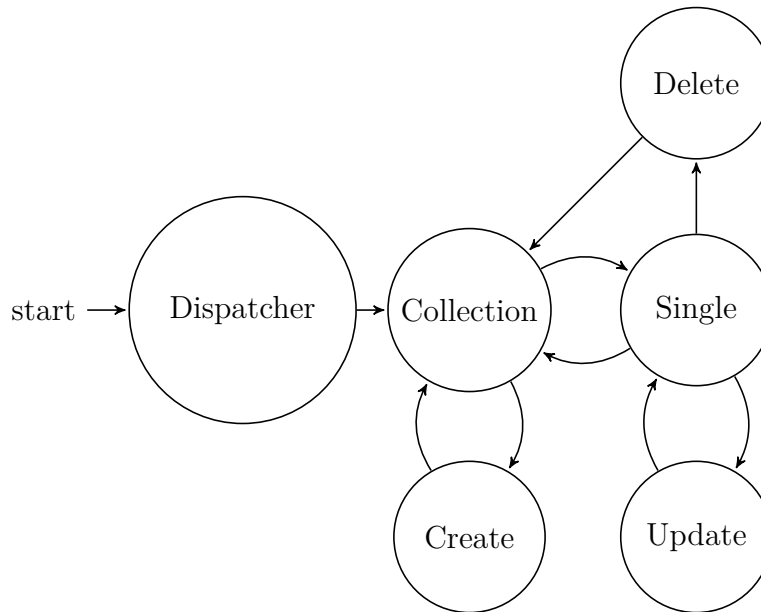


Abbildung 2.1: Aufbau eines *REST-API* mit Hilfe eines *endlichen Automaten*.

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 [13]. 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 heißen 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 verschiedene *Layouts* zur Auswahl. Beispielsweise das *LinearLayout*, mit horizontaler oder vertikaler Orientierung. Ein weiteres Beispiel ist das *RelativeLayout*.

Reichen die Standard-*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, dass 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 *RecyclerView* 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 *CustomView* und Erzeugen einer *CustomView* Klasse.

2.2.1 Registrieren der *CustomView*

Zur Erzeugung und Registrierung von *CustomViews* wird eine Datei *attrs.xml* benötigt. Diese liegt im Ordner *values* im Verzeichnis *res*. In dieser *XML*-Datei werden im *resources*-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.

```
1 <resources>
2     <declare-styleable name="AttributeInput">
3         <attr name="hintText" format="integer"/>
4         <attr name="inputType" format="string"/>
5     </declare-styleable>
6     <declare-styleable name="PersonCardView" />
7 </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*

```

1 <RelativeLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
2   android:id="@+id/relativeLayout"
3   android:layout_width="match_parent"
4   android:layout_height="wrap_content">
5
6   <TextView
7       android:id="@+id/first_name"
8       android:layout_width="wrap_content"
9       android:layout_height="wrap_content"
10      android:text="@string/firstName"/>
11
12   <TextView
13       android:id="@+id/last_name"
14       android:layout_width="wrap_content"
15       android:layout_height="wrap_content"
16       android:text="@string/last_name"/>
17
18 [...]
19
20 </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, 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 dass diese genutzt wird. Die Vererbungsstruktur bringt mit sich, dass die *Konstrukturen* der *CardView* implementiert werden müssen. Die Anzahl dieser *Konstrukturen* hängt von der Minimum *SDK*-Versions des Projekts ab. Dieses Projekt nutzt das Minimum Level 12 weshalb drei *Konstrukturen* überschrieben werden

müssen. Ab einem 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
14     defStyleAttr) {
15     super(context, attrs, defStyleAttr);
16     init(context, attrs, defStyleAttr);
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 die Methode *setPerson* erweitert. Diese ist angelehnt an die Methode *setText* der *TextView*. Sie ermöglicht, dass 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
3     this.firstName.setText(person.getFirstName());
4     this.lastName.setText(person.getLastName());
5     [...]
6 }

```

Mit Hilfe dieser Methode wird die Nutzung 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. Dieses Verfahren wird auch modellgetriebene Softwareentwicklung (MDSD) genannt [15].

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 [10]. 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 *emdded 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]. Anwendungsfälle 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 *acldsl* zu nutzen, sollte er die verwendete *Hostsprache* bereits kennen.

Externe DSLs

Anders als die *internen DSLs* besitzen *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 GEneration von Mobilen Applikationen basierend auf REST Architekturen (GeMARA)

Das Projekt GeMARA beinhaltet eine Reihe von Software-Generatoren, deren Ziel es ist mobile und verteilte Applikationen basierend auf dem *REST* Architekturstil generieren zu lassen. Dafür wurde eine *interne DSL* entwickelt, mit deren Hilfe sowohl *Client*-seitige als auch *Server*-seitige 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 eine *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.

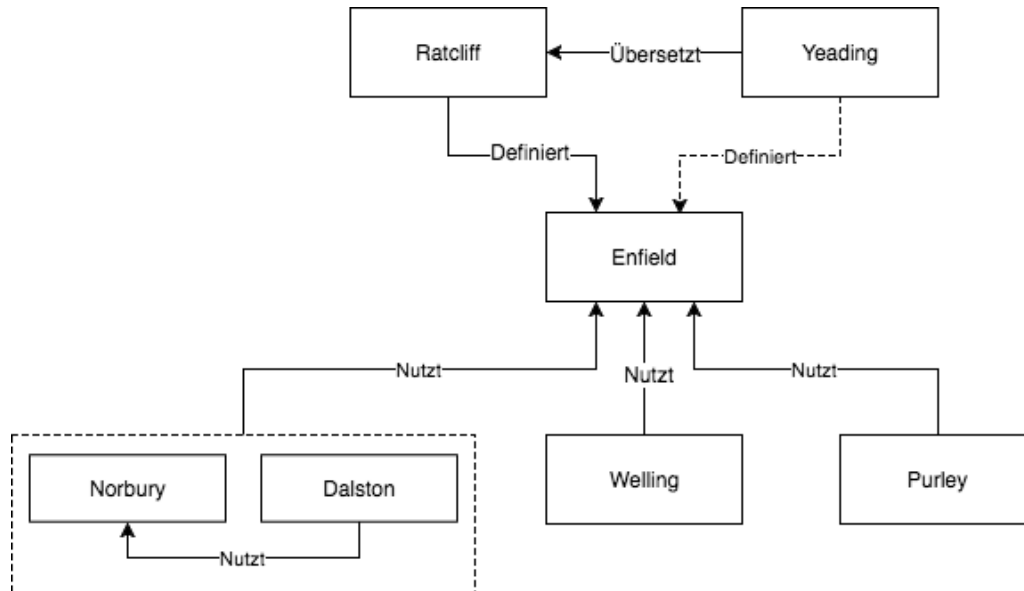


Abbildung 2.2: Aufbau von GeMARA

- **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 nach Ratcliff übersetzt werden.
- **Enfield** liefert das *Meta-Model*, welches für die Beschreibung der gewünschten Applikation 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

In diesem Kapitel wird die Referenzanwendung für diese Arbeit vorgestellt. Zuerst wird das benötigte Backend vorgestellt. Es wird darauf eingegangen, das das benötigte *Application Programming Interface (API)* mit Hilfe von GENERierung von Mobilien Applikationen basierend auf REST Architekturen (GeMARA) generiert wurde und wie das zugehörige *Enfield-Meta-Modell* aussieht. Nach der Vorstellung des Modells wird auch das erzeugte *Application Programming Interface (API)* dargestellt. Anschließend wird die Android Applikation vorgestellt, wobei in diesem Bereich primär auf das *User Interface (UI)* eingegangen wird. Abschließend wird das *Meta-Modell* und mögliche Erweiterungen vorgestellt und gegeneinander abgewogen.

Um einen Generator zu entwickeln, ist es hilfreich, eine Implementierung der gewünschten Applikation mit all ihren Funktionen und Anforderungen zu entwickeln. Bei einer anschließenden Quellcode-Analyse sollte darauf geachtet werden, dass die einzelnen Klassen weitestgehend abstrahiert sind und 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.

3.1 Vorstellung der Referenzanwendung

Die Beispielanwendung soll dem Nutzer die Möglichkeit geben, die Dozenten der Fakultät Informatik der FHWS, und deren Ämter einzusehen, einen neuen Dozenten anzulegen, einen existierenden Dozenten zu bearbeiten oder zu löschen. Neben den Dozenten, soll es weiterhin möglich sein, die Ämter eines Dozenten einzusehen, zu bearbeiten, neu anzulegen oder zu löschen. Für jede dieser Aktionen werden entsprechende Endpunkte in dem *Application Programming Interface (API)* benötigt. Jeder dieser Endpunkte benötigt einen Zugriff auf die darunter liegende Datenbank. Um das zu realisieren, wird ein *Backend-Projekt* mit Hilfe von GeMARA erzeugt.

3.1.1 Backend Referenzimplementierung

Dieses Kapitel stellt die Referenzimplementierung des Backends für die Referenzanwendung vor. Dabei wird das *Enfield-Meta-Modell* und das daraus resultierende *Application Programming Interface (API)* vorgestellt.

***Enfield-Meta-Modell* der Referenzimplementierung**

Dieses Kapitel befasst sich mit dem benötigten *Enfield-Meta-Modell*. Dafür werden Quellcode-Beispiele aufgezeigt und auf deren Besonderheiten eingegangen. In diesem Kapitel wird nicht das ganze Modell vorgestellt, nur die wichtigsten Aspekte daraus. Das komplette Modell kann im Anhang unter Listing 1 eingesehen werden.

In Listing 3.1 wird die Initialisierung des *Enfield-Modells* dargestellt. Zusätzlich werden die Attribute *producerName*, *packagePrefix* und der Name des Projektes festgelegt.

Listing 3.1: Initialisierung des *Enfield-Meta-Modells*.

```

1 public MyEnfieldModel() {
2     this.metaModel = new Model();
3
4     this.metaModel.setProducerName("fhws");
5     this.metaModel.setPackagePrefix("de.fhws.applab.gemara");
6     this.metaModel.setProjectName("Lecturer");
7 }

```

Listing 3.2 zeigt, wie eine Ressource angelegt wird, welche einen Dozenten darstellen soll. Der Ressource wird ein Name zugewiesen und sie bekommt einen *MediaType*. Außerdem werden alle Attribute definiert, in dem sie benannt und einen Datentyp zugewiesen bekommen. Da eines der Attribute eines Dozenten seine Ämter sind, welche als Subressource der *SingleResource Lecturer* dargestellt ist, ist es wichtig, dass das *Enfield-Modell* ebenfalls eine *SingleResource* für diese Ämter besitzt. Dies wird hier durch den Methodenaufwurf *createSingleResourceCharge()* sichergestellt.

Listing 3.2: Erzeugung der *SingleResource Lecturer*.

```

1 this.metaModel.addSingleResource("Lecturer");
2
3 this.lecturerResource = this.metaModel.getSingleResource("Lecturer");
4
5 this.lecturerResource.setModel(this.metaModel);
6 this.lecturerResource.setResourceName("Lecturer");
7 this.lecturerResource.setMediaType(
8     "application/vnd.fhws-lecturer.default+json");
9
10 final SimpleAttribute title = new SimpleAttribute("title",
11     SimpleDatatype.STRING);
12 [...]
13 final SimpleAttribute roomNumber = new SimpleAttribute("roomNumber",
14     SimpleDatatype.STRING);
15 final SimpleAttribute homepage = new SimpleAttribute("homepage",
16     SimpleDatatype.LINK);
17
18 createSingleResourceCharge();
19 final ResourceCollectionAttribute charge = new
20     ResourceCollectionAttribute("chargeUrl", this.chargeResource);
21
22 this.lecturerResource.addAttribute(title);
23 [...]
24 this.lecturerResource.addAttribute(charge);
25
26 addImageAttributeForLecturerResource();

```

Sind alle benötigten Ressourcen im Modell definiert, wird der *endliche Automat* beschrieben, angefangen mit dem *DispatcherState* (3.3). Dieser bekommt einen Namen, und wird als *DispatcherState* dem Modell hinzugefügt.

Listing 3.3: Erzeugung des *DispatcherStates*.

```

1 final GetDispatcherState dispatcherState = new GetDispatcherState();
2 dispatcherState.setName("Dispatcher");
3 dispatcherState.setModel(this.metaModel);
4 this.metaModel.setDispatcherState(dispatcherState);
5 this.dispatcherState = dispatcherState;

```

3 Problemstellung

Beispielhaft für alle nachfolgenden *States* zeigt Listing 3.4 wie der *State GetAllLecturers* erzeugt wird. Auch dieser *State* bekommt einen Namen, daneben wird ihm die Ressource zugewiesen, welche er bedienen soll. Neben diesen Eigenschaften, werden dem *State* alle *Transitionen* hinzugefügt. In diesem Fall wird zusätzlich dem *DispatcherState* die Information übergeben, dass der *GetAllLecturers State* sein *Folgestate* ist. Alle weiteren *States* werden analog definiert (siehe Listing 1).

Listing 3.4: Erzeugung des *GetAllLecturers States*.

```
1 final GetPrimaryCollectionResourceByQueryState
   getAllLecturersCollectionState = new
   GetPrimaryCollectionResourceByQueryState();
2 getAllLecturersCollectionState.setName("GetAllLecturers");
3 getAllLecturersCollectionState.setModel(this.metaModel);
4 getAllLecturersCollectionState.setResourceType(this.lecturerResource);
5
6 this.dispatcherState.addTransition(new
   ActionTransition(getAllLecturersCollectionState, "getAllLecturers"));
7 getAllLecturersCollectionState.addTransition(this.getLecturerByIdState);
8
9 this.metaModel.addState(getAllLecturersCollectionState.getName(),
   getAllLecturersCollectionState);
10
11 this.getCollectionOfLecturersState = getAllLecturersCollectionState;
```

Vorstellung des *Application Programming Interface (API)*

Mit Hilfe des zuvor beschriebenen *Enfield-Meta-Modells* wird ein *Application Programming Interface* generiert, welches in diesem Kapitel vorgestellt wird. Die Abbildung 3.1 zeigt das *Application Programming Interface*, welches für die Beispielanwendung benötigt wird. Dieses *API* entspricht einem *endlichen Automaten* und spiegelt alle Funktionen der Applikation wieder.

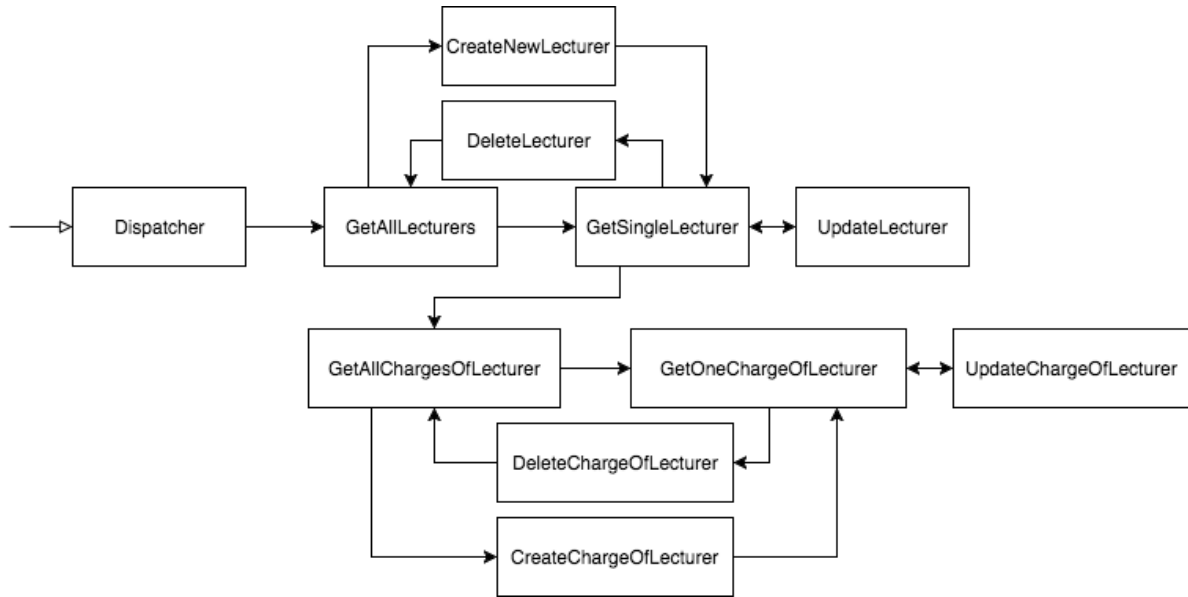


Abbildung 3.1: Darstellung des *API* der Beispielanwendung.

3.1.2 Android Referenzimplementierung

Nach der Vorstellung des Backends, geht dieses Kapitel darauf ein, wie Informationen des *Application Programming Interfaces* in die Android Applikation einfließen. Für die Realisierung der gewünschten Funktion werden folgende *Views* benötigt: eine *RecyclerView*, welche alle Dozenten in jeweils einer eigenen *CardView* darstellen; eine *DetailView* - diese zeigt einen einzelnen Dozenten und all seine Informationen; jeweils eine *InputView* zur Erzeugung eines neuen Dozenten beziehungsweise zum Bearbeiten eines existierenden Dozenten, eine *RecyclerView*, welche die Ämter eines Dozenten anzeigt, die *DetailView* eines Amtes sowie wiederum jeweils eine *View* zum Bearbeiten beziehungsweise zur Neuanlage eines Amtes.

***User Interface (UI)* der Referenzimplementierung**

In diesem Kapitel wird das *User Interface* der Android Applikation vorgestellt. Dabei wird gezeigt, wie die einzelnen *Views* umgesetzt werden. Wie dem *Application Programming Interface* entnommen werden kann, steigt der Nutzer mit der Liste aller Dozenten in die Applikation ein. Diese Liste ist in diesem Fall wie in der Einleitung beschrieben mit einer *RecyclerView* und einzelnen *CardViews* realisiert. Diese Liste kann in Abbildung 3.2 eingesehen werden.

3 Problemstellung

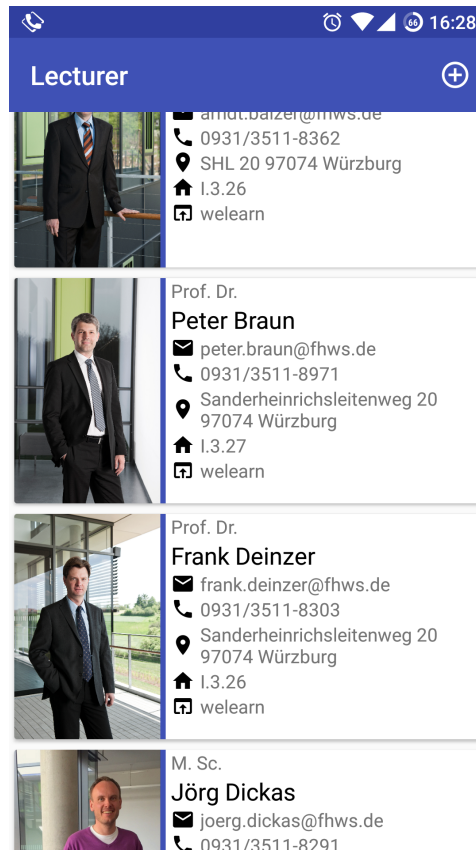
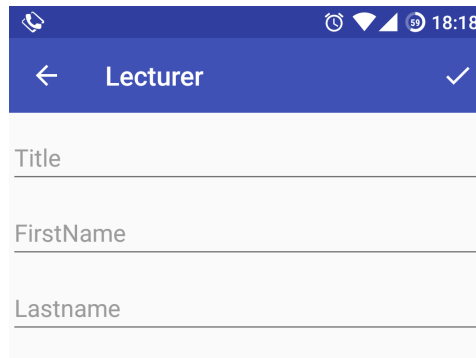


Abbildung 3.2: *RecyclerView* zur Darstellung aller Dozenten.

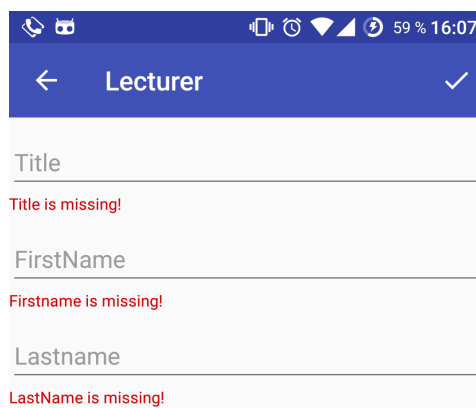
Über den *Plus-Button* links oben in der *View* kommt der Nutzer zu der *View*, welche es ermöglicht, einen neuen Dozenten anzulegen. Diese *View* besteht aus *EditText*-Feldern, welche Vorgeben, welche Informationen zur Neuanlage benötigt werden. Die Abbildung 3.3 zeigt diese *View*. Diese *View* validiert auch, ob eine Eingabe getätigt wurde, andernfalls wird eine Fehlermeldung angezeigt. Diese Darstellung der Fehlermeldung ist in Abbildung 3.4 dargestellt.

3 Problemstellung



The screenshot shows a mobile application interface for creating a lecturer. At the top, there is a blue header bar with a back arrow, the title 'Lecturer', and a checkmark icon. Below the header, there are three text input fields labeled 'Title', 'FirstName', and 'LastName'. The status bar at the very top indicates the time is 18:18 and the battery level is 59%.

Abbildung 3.3: Ausschnitt der *View* zur Erstellung eines Dozenten.



This screenshot shows the same 'Lecturer' form as in the previous image, but with red error messages displayed below each input field. The 'Title' field has the message 'Title is missing!', the 'FirstName' field has 'Firstname is missing!', and the 'LastName' field has 'LastName is missing!'. The status bar at the top shows the time as 16:07 and the battery level as 59%.

Abbildung 3.4: Fehlermeldung bei der Neuanlage eines Dozenten.

Durch die Neuanlage eines Dozenten oder durch den Klick auf seine Karte in der Liste, wird der Nutzer auf die Detailansicht eines Dozenten weitergeleitet (Abbildung 3.5). Hier bekommt der Nutzer die Möglichkeit detaillierte Informationen zum betroffenen Dozenten zu bekommen. Des Weiteren bekommt er die Möglichkeit, den aktuellen Dozenten zu bearbeiten oder diesen zu löschen. Diese Aktionen können über das Kontextmenü aufgerufen werden. Wobei die *View* zum Editieren des Dozenten analog der *View* zur Neuanlage aussieht mit der Ausnahme, dass die vorhandenen Daten bereits vor ausgefüllt sind. Das Löschen des Dozenten wird über einen *Dialog* realisiert. Dieser *Dialog* ist in Abbildung 3.6 abgebildet.

3 Problemstellung

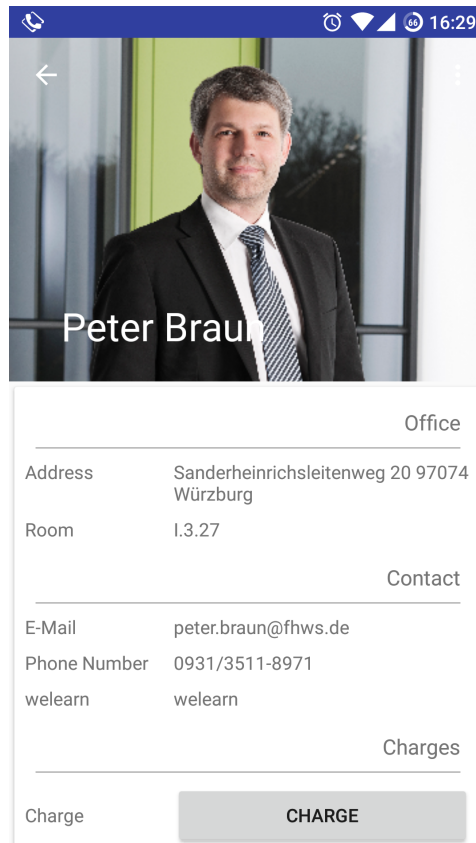


Abbildung 3.5: Detailansicht eines Dozenten.

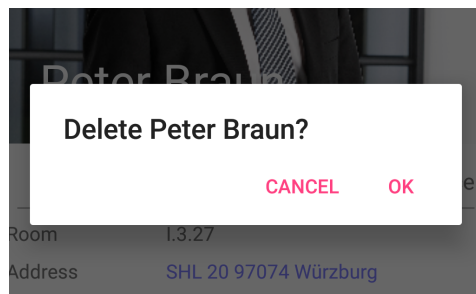


Abbildung 3.6: *Dialog* zum Löschen eines Dozenten.

Über den *Charge-Button* gelangt der Nutzer zur Liste mit den Ämtern des Dozenten. Die *Views* für diese Ämter sind analog zu denen der Dozenten. Mit der Ausnahme, dass bei Neuanlage, beziehungsweise beim Bearbeiten eines Amtes dieses mal nicht ausschließlich *EditText* zur Verfügung steht. Da die Ämter die zwei Datumsattribute für den Start und das Ende besitzen, wurde hierfür das *DateTimePicker-Widget* eingebaut. Dieses kann in Abbildung 3.7 eingesehen werden.

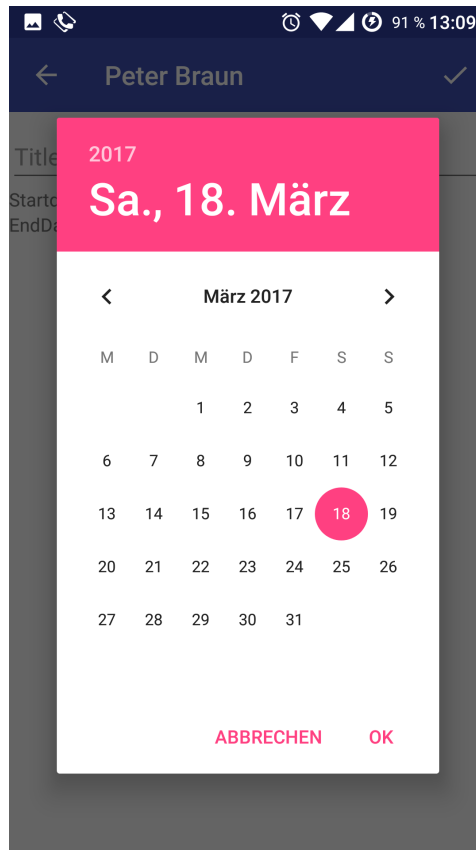


Abbildung 3.7: *DateTimePicker-Widget* zur Datumsauswahl.

Zahlen und Fakten

In diesem Abschnitt sollen Statistiken der Referenzimplementierung vorgestellt werden. So lässt sich die Applikation in zwei Bereiche einteilen: die Applikation an sich, diese besitzt den kompletten spezifischen Code und eine Bibliothek. Diese beinhaltet den kompletten generischen Code sowie die *CustomViews*. Die gesamte Anwendung besitzt ungefähr 3000 *Lines of Java Code* und circa 1000 Zeilen an *XML Code*.

Die Applikation ist der kleinere Teil der Implementierung, sie enthält elf Java Klassen, wobei es sich dabei um vier *Activities* und sieben *Fragments* handelt. Daneben besitzt sie sechs *Layout-XML*-Dateien und zwei *Animations-XML*-Dateien. Daneben noch die üblichen *XML*- und *Gradle*-Dateien.

Die Bibliothek ist mit 44 Java Klassen wesentlich größer, wobei hiervon 15 Klassen spezifischen Code enthalten. Die Klassen lassen sich in *abstrakte Activities*, *abstrakte Fragments*, *abstrakte Adapter*, *abstrakte CustomViews*, *abstrakte Models*, *abstrakte View-*

holder, Klassen für die Netzwerkkommunikation, *Adapter*, *CustomViews*, *Models* und *ViewHolder* einteilen. Neben den Java Klassen besitzt die Bibliothek 14 Layout- und drei Menü-Klassen. Diese 17 Klassen sind alles *XML*-Dateien. Auch die Bibliothek besitzt die weiteren, üblichen *XML*- und *Gradle*-Dateien.

3.2 Analyse der Android Anwendung

Dieses Kapitel beschäftigt sich mit der Android Referenzimplementierung. Es wird der Aufbau der Applikation vorgestellt und anhand dessen analysiert, welcher Programmcode als *Plattformcode* und welcher Programmcode generiert werden muss. Es werden auch ein paar Kennzahlen vorgestellt, um ein besseres Bild der Komplexität der Applikation zu erhalten. Anschließend werden die einzelnen *Views* analysiert. Diese Analyse beschäftigt sich mit dem Design und den Funktionen der *Views*.

3.2.1 Analyse des Aufbaus

Wird der Aufbau der Referenzimplementierung analysiert, so fällt auf, dass es möglich ist die meisten Klassen soweit zu abstrahieren, dass diese keine Projekt spezifischen Informationen mehr enthalten. Im Falle dieser Referenzimplementierung keine Informationen zu Dozenten oder deren Ämter. Diese Klassen ohne diese spezifischen Informationen, wird im laufenden als *Plattformcode* oder generischer Code bezeichnet. Das Ziel bei der Referenzimplementierung ist, möglichst viel *Plattformcode* und möglichst wenig spezifischen Code in der Anwendung zu haben.

Das Schaubild 3.8 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.

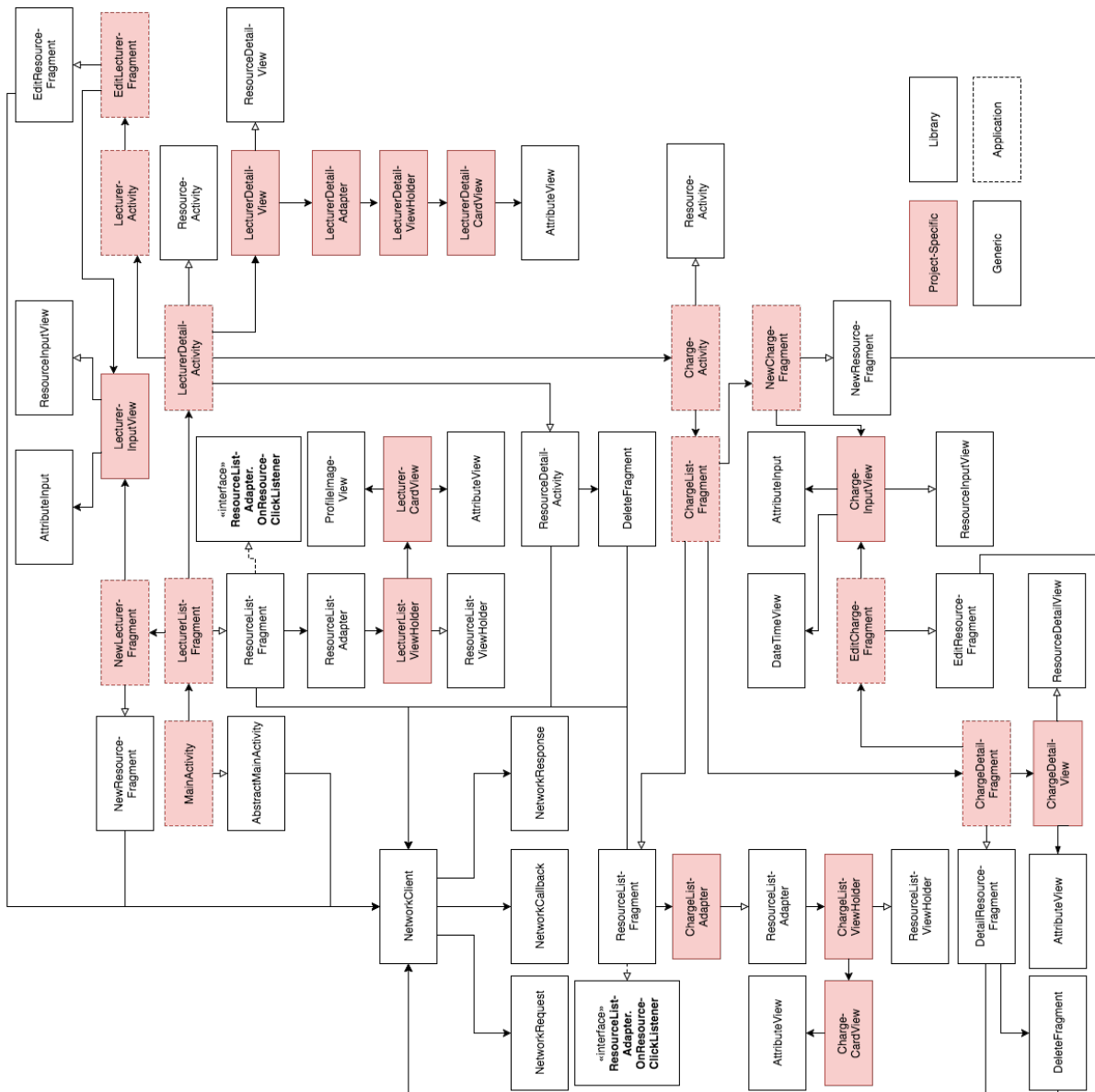


Abbildung 3.8: Aufbau der Referenzimplementierung.

Daneben zeigt die Abbildung 3.8 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 *Activities*, 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 mit wenigen Zeilen Programmcode verwendet werden.

Diese Art der Aufteilung ermöglicht, dass ein Applikations 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.

3.2.2 Analyse der Android Views

In diesem Kapitel sollen die *Views* der Android Applikation an sich analysiert werden. Hierfür werden die Bereiche: Aufbau der *Views*, Darstellung von Schrift, und Aktionen bei Klick genauer betrachtet. Um ein *Meta-Modell* für Android Anwendungen zu entwickeln, muss der Designer untersuchen, welche Eigenschaften dieses Modell besitzen soll. Diese Eigenschaften spiegeln die Möglichkeiten wider, die Android Anwendung zu beschreiben. Für das Extrahieren dieser Eigenschaften ist ein guter Ansatz, eine Referenzimplementierung, zu entwickeln. Diese Referenz dient fortan als Beispiel. Weiterhin stellt sie das erste zu erreichende Ziel dar. Alle Bemühungen sollten darauf hinauslaufen, eine Applikation generieren zu lassen, welche der Referenzimplementierung gleicht.

Schon beim Entwickeln der Referenz muss sich der Entwickler Gedanken darüber machen, welche *Views* die Anwendung besitzen soll. Diese *Views* entscheiden auch über die Funktionalitäten, welche der Entwickler den Nutzern zur Verfügung stellen will. So wird bereits bei der Planung und Entwicklung der Applikation beschrieben welche Features realisiert werden. Dieser Funktionsumfang beschreibt ob der Nutzer Listen- und Detailansichten zur Verfügung hat und ob er Datensätze löschen, neu anlegen beziehungsweise bearbeiten darf. Mit der Entscheidung, dass es eine Möglichkeit zur Neuanlage und Bearbeitung von Datensätzen geben soll, muss zusätzlich festgelegt werden, welche Attribute des Datensatzes bearbeitet werden dürfen und welche minimal notwendig sind.

Ist bekannt welche *Views* realisiert werden, muss über den Aufbau der einzelnen *Views* entschieden werden. Es müssen Entscheidungen über die Anordnung der darzustellenden Informationen innerhalb einer *View* getroffen werden. Diese Entscheidungen beinhalten neben der Strukturierung und Darstellung textueller Informationen auch Überlegungen zum Erscheinungsbild. Zum Erscheinungsbild gehören Eigenschaften wie Schriftgröße oder Schriftfarbe. Ist eine *View* fertig designed, steht fest, in welcher Reihenfolge gegebene Informationen angezeigt werden. Ob Informationen wie Vorname und Nachname zusammengefasst werden. Auch hat der Entwickler entschieden, ob alle existierenden Daten in der entsprechenden Ansicht relevant sind oder ob darauf verzichtet werden kann. Auch ist klar wo und wie möglicherweise existierende Bilder dargestellt werden sollen.

Neben diesen auf das *User Interface (UI)* bezogenen Kriterien müssen auch Entscheidungen darüber gefällt werden, ob diese angezeigten Informationen ausschließlich informative Details sind oder ob diese interaktiv sind. Das heißt der Benutzer der Android Applikation soll die Möglichkeit haben, weitere Funktionen durch das Anklicken einer

dieser Felder auszuführen. Mögliche Aktionen wären beispielsweise das Öffnen der Anwendung *Maps* beim Klick auf eine Adresse oder das Öffnen eines Webbrowsers beim Anklicken eines Hyperlinks.

All diese Entscheidungen, welche über den Aufbau und das Design der Android Applikation entscheiden, sind für einen Generator wichtig. Dieser benötigt all diese Informationen, um diese in der zu generierende Anwendung zu realisieren. Hierfür muss ein *Meta-Modell* entwickelt werden, welches alle der oben genannten Beschreibungen im Bezug zur Android Applikation widerspiegelt. Das Modell muss alle Informationen über die Anzahl und Arten der *Views*, deren Aufbau und die exakte Darstellung von Schrift, Bildern und möglichen Funktionen, welche bei Klick ausgeführt werden sollen, besitzen.

3.3 *Meta-Modell*

Nach dem die Referenzimplementierung vorstellt und analysiert wurde, wurden alle relevanten Informationen erkannt und zusammengestellt. Diese Zusammenstellung an Daten, welche die Applikation beschreiben, wird *Meta-Modell* genannt.

3.3.1 Kompatibilität mit GeMARA und andern möglichen Clients

Da Enfield primär für die Generierung von Anwendungen im *Backend*-Bereich entwickelt wurde, in welchem die Gestaltung von *User Interfaces (UI)* eine eher untergeordnete Rolle spielen, muss die Erweiterung auch dieses Feature realisieren. Neben all diesen Erweiterungen muss auch sichergestellt werden, dass das *Meta-Modell* auch weiterhin für das Generieren von *Backends* genutzt werden kann. Idealerweise, ohne die Überarbeitung der bereits entwickelten Software-Generatoren.

Die Abbildung 3.9 zeigt die vereinfachte Modell-Klasse des *Enfield-Meta-Modells*. In dieser Klasse sind bereits die wichtigsten Informationen wie zum Beispiel der Name der Applikation oder unter welchem *Package* diese zu finden ist, vorhanden. Neben diesen grundsätzlichen Informationen liefert die Modell-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. Jeder dieser folgenden *States*, besitzt wiederum eine Collection mit *Transitionen*, welche auf die nachfolgenden *States* verweisen. So wird mit Hilfe der *Transitionen* und der *States* der *endliche Automat* beschrieben. Der Generator kann diese Beschreibung nutzen, um zu entscheiden in welcher Reihenfolge, welche Klassen generiert werden müssen.

3 Problemstellung

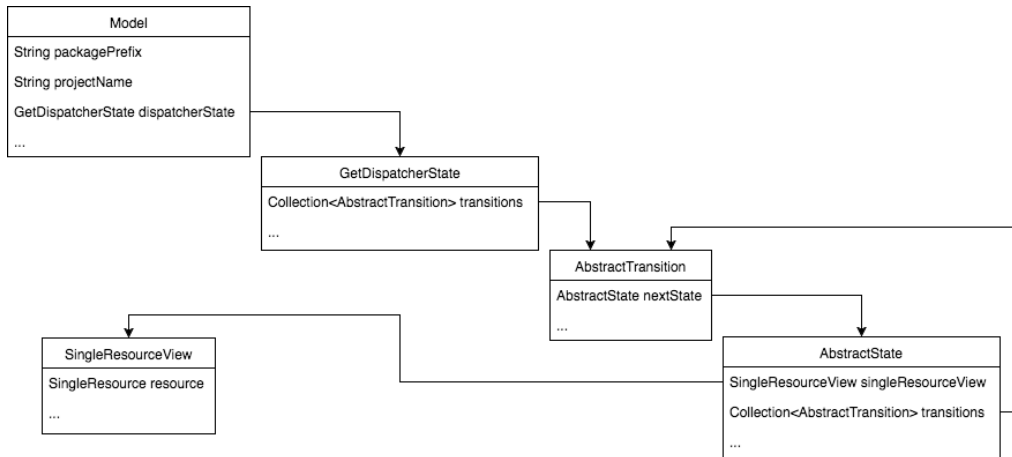


Abbildung 3.9: Vereinfachter Aufbau des *Enfield-Meta-Modells*.

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

3.3.2 Eigenes *Android-Meta-Modell*

Es besteht die Möglichkeit, die Modell-Klasse um ein Attribut *Android-Meta-Modell* zu erweitern. Die Abbildung 3.10 zeigt schemenhaft ein Beispiel, wie ein *Android-Meta-Modell* aussehen könnte. Auffällig hierbei ist, dass viele Informationen, die das *Enfield-Modell* bereits liefern würde, noch einmal explizit beschrieben werden müssen. Ein Beispiel wären die *Transitionen*, zwischen den *Fragments* beziehungsweise zwischen den *Activities*.

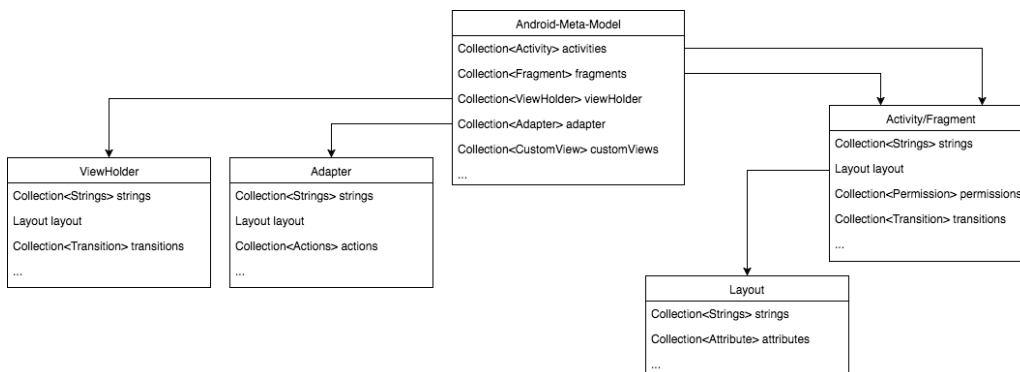


Abbildung 3.10: Möglicher Aufbau eines *Android-Meta-Modells*.

Der Nutzer des Software-Generators, muss ziemlich viel über den Ablauf und die Funktionsweise einer Android-Anwendung wissen, um diesen Generator sinnvoll verwenden zu können. Dabei bleibt zusätzlich noch die Möglichkeit, dass der Nutzer eigens geschriebene Methoden in das Modell einpflegen kann. John Abou-Jaoudeh et al., haben in ihrer Arbeit ein *Meta-Modell* entwickelt, welches genau solche Features unterstützt[1].

Der Vorteil einer solchen Erweiterung des *Enfield-Modells* ist, dass 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-Modell* und einmal im *Enfield-Meta-Modell*. Bei jeder Änderung gilt dies zu berücksichtigen. Der nächste Nachteil ist, dass der Nutzer des Software-Generators sich in der Entwicklung von Android Anwendungen auskennen muss. Er muss genau das Zusammenspiel von *ViewHolders*, *Adapttern*, *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 Modells auf Android. Wird das *Enfield-Modell* um ein *Android-Meta-Modell* 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-Modell* mit einem entsprechenden *Meta-Modell* erweitert werden.

3.3.3 Allgemeine Erweiterungen des *Enfield-Modells*

In dieser Arbeit wurde sich für die Variante entschieden, das *Enfield-Modell* an geeigneter Stelle zu erweitern. Diese Stelle befindet sich in den einzelnen *States*. Jede Instanz des *AbstractState* besitzt ein Attribut *SingleResourceView*. Diese Klasse wird um die Attribute, welche benötigt werden, erweitert. In der Abbildung 3.11 ist der vereinfachte Aufbau des *AbstractStates* und einer *SingleResourceView* zu sehen.

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

3 Problemstellung

betreffen. Hierfür muss das *Enfield-Modell* noch an einer anderen Stelle erweitert werden. Es erscheint sinnvoll die Erweiterung direkt in der Modell-Klasse vorzunehmen. So kann der Generator schon am Anfang auf diese Daten zugreifen und diese verarbeiten.

Die Abbildung 3.11 zeigt das *Enfield-Modell*, welches um die oben genannten Informationen erweitert wurde.

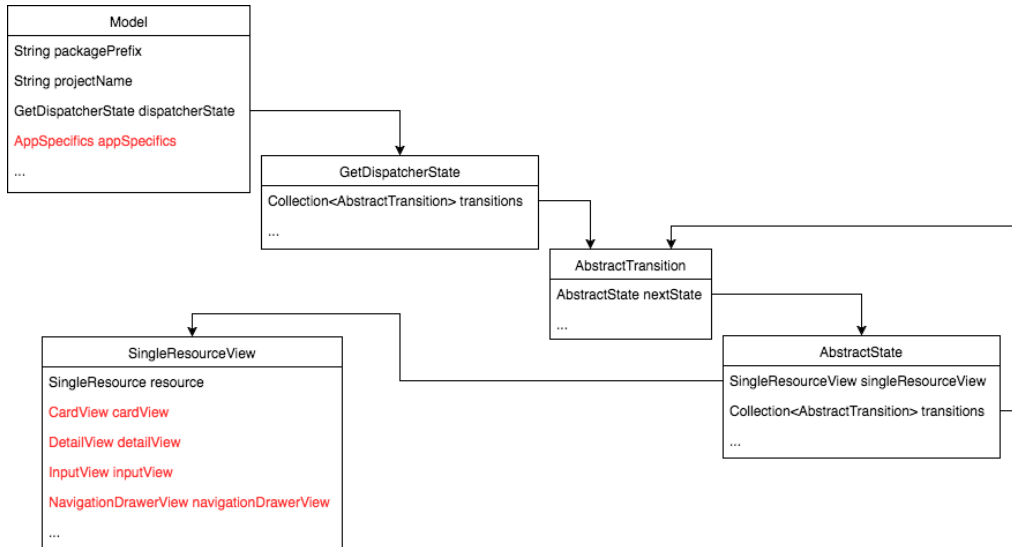


Abbildung 3.11: Vereinfachter Aufbau des erweiterten *Enfield-Meta-Modells*.

Der Nachteil dieser Methode ist, dass 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 Modell-Klasse nötig. Die Vorteile wurden jedoch zuvor, in Abschnitt 3.3.3, schon einmal erwähnt. Der Generator kann das Modell 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 gestaltet ist.

3.3.4 Analyse der benötigten Dateien für das *Meta-Modell*

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

3 Problemstellung

Die Analyse in dieser Arbeit beschränkt sich auf die *Clients* Android und Polymer-Webkomponente. Bei beiden wird das User Interface (UI) nach den Richtlinien, des von Google entwickelten Material Design, erstellt [12]. Diese Richtlinien schreiben bereits viele nötigen Informationen für die Oberflächengestaltung vor. So wird beispielsweise definiert, dass Einträge in einer Liste, als Karte dargestellt werden sollen. Abstände und Icons werden ebenfalls festgelegt.

CardView

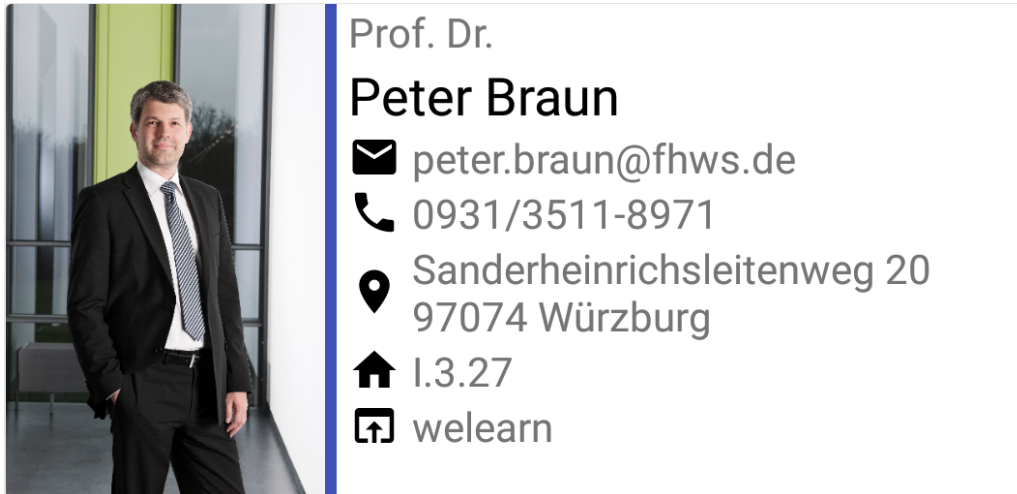


Abbildung 3.12: Beispiel einer *CardView* eines Dozenten nach Material Design.

Listing 3.5: Demo Daten eines Dozenten.

```

1  [...]
2  {
3    "address": "Sanderheinrichsleitenweg 20 97074 Wuerzburg",
4    "chargeUrl": {
5      "href": "https://apistaging.fiw.fhws.de/mig/api/lecturers/4/charges",
6      "rel": "chargeUrl",
7      "type": "application/vnd.fhws-charge.default+json"
8    },
9    "email": "peter.braun@fhws.de",
10   "firstName": "Peter",
11   "homepage": {
12     "href": "http://www.welearn.de/[...]/prof-dr-peter-braun.html",
13     "rel": "homepage",
14     "type": "text/html"
15   },
16   "id": 4,
17   "lastName": "Braun",
18   "phone": "0931/3511-8971",
19   "profileImageUrl": {
20     "href": "https://apistaging.fiw.fhws.de/[...]/4/profileimage",
21     "rel": "profileImageUrl",
22     "type": "image/png"
23   },
24   "roomNumber": "I.3.27",
25   "self": {
26     "href": "https://apistaging.fiw.fhws.de/mig/api/lecturers/4",
27     "rel": "self",
28     "type": "application/vnd.fhws-lecturer.default+json"
29   },
30   "title": "Prof. Dr."
31 }
32 [...]

```

Die *JavaScript Object Notation (JSON)* Repräsentation unter Listing 3.5 beschreibt das Beispiel aus Abbildung 3.12. 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 in der Liste für jeden einzelnen Dozenten angezeigt werden sollten. Ist es sinnvoll 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 auf der Karte bestimmt werden. Neben der Reihenfolge gibt es noch die Möglichkeit, Schriftgröße oder Schriftfarbe der

3 Problemstellung

einzelnen Attribute unterschiedlich zu gestalten. Auch müssen die Standardicons den jeweiligen Attributen zugewiesen werden. Es sollte zudem möglich sein, einzelnen Attributen bestimmte Aktionen zuzuweisen. Beispielsweise beim Klick auf eine Homepage, sollte diese im Browser geöffnet werden, oder beim Klick auf die Adresse sollte sich die Applikation *Maps* öffnen und die angeklickte Adresse dort anzeigen. Bei einem Attribut mit Hyperlink zu einer Website sollte es möglich sein, einen mitgegebenen Text anstelle des Hyperlinks anzuzeigen.

Besitzt die Karte ein Bild, so sollte der Nutzer die Möglichkeit besitzen, zu entscheiden, ob dieses auf der linken oder rechten Seite der Karte dargestellt werden soll.

DetailView

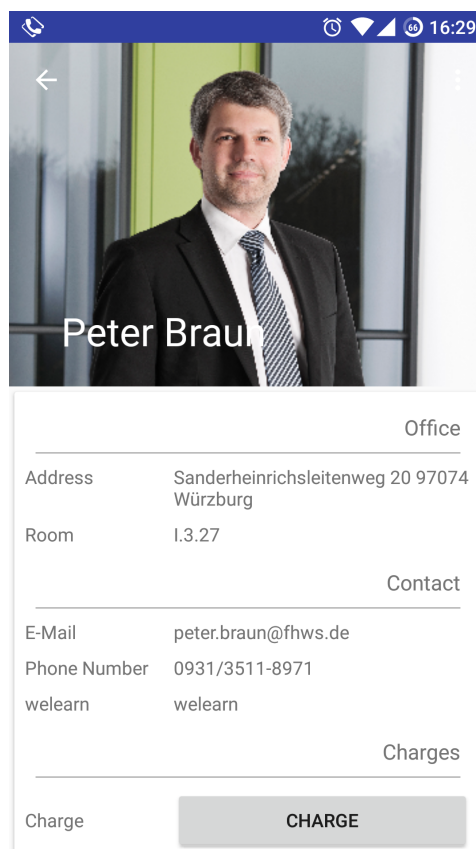


Abbildung 3.13: Beispiel einer *DetailView* eines Dozenten nach Material Design.

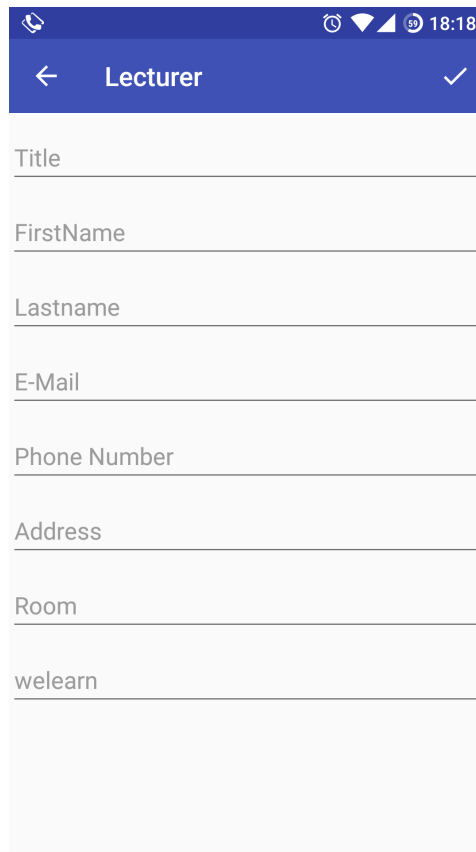
Die zur Verfügung stehenden Daten sind die, die unter Listing 3.5 einzusehen sind.

3 Problemstellung

Analog zu der *CardView* stellt sich auch bei der *DetailView* die Frage, welche Daten dargestellt werden sollen. Hier jedoch gibt es zusätzlich zu der horizontalen Gruppierung (Beispiel: Vornamen und Nachnamen) auch noch eine vertikale Gruppierung. Diese wird im Weiteren auch Kategorisierung genannt. In der detaillierten Ansicht eines Dozenten gibt es die Möglichkeit, Attribute zu kategorisieren und jede 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 *CollapsingToolbar* dargestellt [5].

InputView



The screenshot shows a mobile application interface for creating a new lecturer. At the top is a blue header bar containing a back arrow on the left, the word 'Lecturer' in the center, and a checkmark on the right. Below the header, there is a vertical list of text input fields. The labels for these fields are: 'Title', 'FirstName', 'Lastname', 'E-Mail', 'Phone Number', 'Address', 'Room', and 'welearn'. The 'welearn' field is the last one shown, with a large empty space below it.

Abbildung 3.14: Beispiel einer *View* zum Anlegen eines Dozenten.

Für das Anlegen eines Dozenten oder auch zum Bearbeiten muss entschieden werden, welche Attribute zum Anlegen nötig sind. Auch hier ist es notwendig die Reihenfolge zu bestimmen. Jedoch kommen in dieser *InputView* für jedes Attribut noch die Möglichkeit hinzu einen *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 Eingebene nicht dem Erwarteten entspricht. Zum Beispiel wurde in das Feld für die E-Mail eine Telefonnummer eingegeben. Oder es wurde ein regulärer Ausdruck mitgegeben und das Eingebene entspricht nicht dessen Anforderungen.

Programmablauf und Klick-Aktionen

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

Auch die Aktionen, welche durch einen Klick auf ein bestimmtes Attribut ausgeführt werden sollen, beschränken sich auf Android Standardaktionen. Beispielsweise das Wechseln zu den *Maps*, zu einem *E-Mail Client*, dem *Browser* oder zum *Anrufsmenü*. Jede dieser Aktion ergibt sich aus den Typen der Attribute, weswegen diese auch nicht weiter definiert werden müssen.

3.3.5 Design der *View-Meta-Modelle*

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

3 Problemstellung

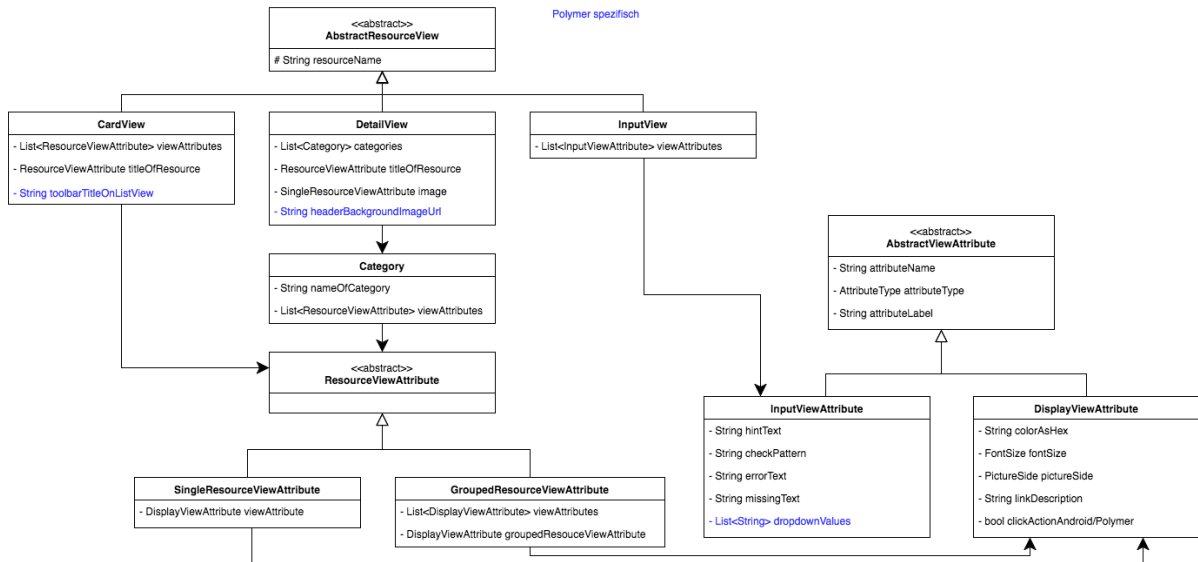


Abbildung 3.15: Aufbau der Views zur Erweiterung des *Enfield-Modells*.

Die Abbildung 3.15 zeigt den Aufbau der Objekte, mit welchem das *Enfield-Modell* erweitert wird. Die drei Views: *CardView*, *DetailView* und *InputView* sind alles Instanzen von *AbstraktResourceView*. Jede der Views weiß, durch die Zuordnung mit Hilfe des Ressourcennamens, welche Ressource sie darstellen soll. 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* zu den anzeigenden Views und die *InputView* zur zweiten Kategorie.

Anzeigende Views

Diese View-Typen haben die Aufgabe eine Liste aller 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 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 Attribut *image*, dieses wird hier aus der Liste der Attribute herausgezogen, da dieses Attribut bestimmt, ob die View eine *CollapsingToolbar* besitzen wird oder nicht. Wiederum haben beide Views das Attribut *titleOfResource*. Dieses bestimmt, welches Attribut unserer Ressource beispielsweise in der *Toolbar* angezeigt wird.

3 Problemstellung

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

Mit Hilfe der Listen, den Titelattributen und dem Bildattribut kann das Erscheinungsbild einer *View* gut beschrieben werden. Als nächstes wird auf Möglichkeit, Schriftgrößen, Schriftfarben und Klick-Aktionen zu definieren eingegangen. 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 Ausprägungsarten: *SingleResourceViewAttribute* und *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*.

Während bei der *SingleResourceViewAttribute* das *GroupedResourceViewAttribute* wiederum gibt eine Liste von diesen *DisplayViewAttributen*, welche dann die anzuzeigenden Informationen widerspiegeln. Weitergehend besitzt das *GroupedResourceViewAttribute* auch noch ein *DisplayViewAttribute*, welches die neu entstandene Gruppierung beschreiben soll.

Ein *DisplayViewAttribute* besitzt nun die Möglichkeit, Schriftgröße und -farbe zu definieren. Die angegebene Farbe muss in hexadezimaler Darstellung angegeben werden, wird keine Farbe mitgegeben, wird die Defaultfarbe der Anwendung genommen. In der Regel ist diese Schwarz. Die Schriftgröße wiederum ist auf drei Stufen beschränkt. Es gibt die Möglichkeit den Text in klein, normal und groß darzustellen. Per default ist normal eingestellt. Wegen der Oberklasse *AbstractViewAttribute* besitzt das *DisplayViewAttribute* noch das Attribut *attributeName*, dieses muss exakt so heißen wie in der Definition der Ressource. Mit dem *attributeLabel* kann angegeben werden, wie dieses Attribut in der *View* angezeigt werden soll. Die Abbildung 3.13 zeigt die Verwendung von den Labels Beispielsweise von der E-Mailadresse des Dozenten steht *E-Mail*, dieser String entspricht dem Label des Attributes. Weiterhin muss angegeben werden von welchem Typ das aktuell beschriebene Attribut ist. Dies geschieht 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 wird oder 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 beispielsweise bei einem

3 Problemstellung

URL-Attribut noch eine Beschreibung mitgegeben werden, welche anstelle der Hyperlinks angezeigt werden soll. Bei einem Bild kann beispielsweise noch bestimmt werden, ob dieses links oder rechts dargestellt werden soll.

Nachfolgend wird auf einige Besonderheiten der Nutzung der eingebundenen *Views* eingegangen und diese genauer erklärt. So zeigt Listing 3.6 beispielsweise das Erzeugen eines *GroupedResourceViewAttributes*. Hierfür werden erst einmal drei *DisplayViewAttribute* definiert. Das erste beschreibt hierbei das Aussehen, den Namen und den Typ der Gruppierung. Die Gruppe in diesem Beispiel wird aus den beiden Attributen *firstName* und *lastName* zusammengesetzt. Beide Attribute sind vom Typ *TEXT*; auch die Gruppe wird von diesem Typ sein. Der String im Konstruktor ist der Name dieses Attributes. Stellt das Attribut ein Attribut aus der Ressource dar, wie der *firstName* beziehungsweise *lastName*, muss dieser Name identisch mit dem Attribut der Ressource sein. Neben der Zusammensetzung der Gruppe wird hier ebenfalls definiert, wie dies dargestellt werden soll. Mit *setFontSize(DisplayViewAttribute.FontSize.LARGE)* wird deklariert, dass die Gruppen mit einer großen Schriftgröße dargestellt werden, und die Methode *setFontColor("#000")* bestimmt, dass die Schrift schwarz ist. Bei einer Gruppe hat es kein Effekt, wenn die Schriftfarbe oder Größe der einzelnen Gruppenmitglieder bestimmt wird. Die Darstellung ist einzig von den Attributen der Gruppe abhängig.

Listing 3.6: Erstellung eines *GroupedResourceViewAttributes*.

```
1 DisplayViewAttribute nameAttribute = new DisplayViewAttribute("name",
   ViewAttribute.AttributeType.TEXT);
2 nameAttribute.setFontSize(DisplayViewAttribute.FontSize.LARGE);
3 List<DisplayViewAttribute> nameAttributes = new ArrayList<>();
4
5 DisplayViewAttribute firstNameAttributes = new
   DisplayViewAttribute("firstName", ViewAttribute.AttributeType.TEXT);
6 firstNameAttributes.setAttributeLabel("FirstName");
7 nameAttributes.add(firstNameAttributes);
8
9 DisplayViewAttribute lastNameAttributes = new
   DisplayViewAttribute("lastName", ViewAttribute.AttributeType.TEXT);
10 lastNameAttributes.setAttributeLabel("LastName");
11 nameAttributes.add(lastNameAttributes);
12
13 GroupResourceViewAttribute name;
14 try {
15 nameAttribute.setFontColor("#000");
16 name = new GroupResourceViewAttribute(nameAttribute, nameAttributes);
17 } catch (DisplayViewException ex) {
18 name = null;
19 }
```

3 Problemstellung

Das Listing 3.7 beschreibt die Definition einer *Category* als Teil einer *DetailView*. In diesem Listing wird eine *Category* mit dem Namen *Office* erzeugt. Diese Kategorie besitzt zwei Attribute, welche als *DisplayViewAttribute* dargestellt werden. Eines der beiden Attribute ist in diesem Fall die Adresse. Es wird definiert, dass dieses *DisplayViewAttribute* vom Type *LOCATION* ist und dass es eine Aktion beim Anklicken geben soll. Des Weiteren wird definiert, dass dieses Attribut ein Label *Address* besitzt.

Listing 3.7: Erstellung einer *Category*.

```
1 [...]
2 new Category("Office", getOfficeResourceViewAttributes());
3 [...]
4
5 private static List<ResourceViewAttribute>
6     getOfficeResourceViewAttributes() {
7
8     DisplayViewAttribute addressAttribute = new
9         DisplayViewAttribute("address",
10             ViewAttribute.AttributeType.LOCATION);
11     addressAttribute.setAttributeLabel("Address");
12     addressAttribute.setClickActionAndroid(true);
13     SingleResourceViewAttribute address = new
14         SingleResourceViewAttribute(addressAttribute);
15     officeAttributes.add(address);
16
17     DisplayViewAttribute roomAttribute = new
18         DisplayViewAttribute("roomNumber", ViewAttribute.AttributeType.TEXT);
19     roomAttribute.setAttributeLabel("Room");
20     roomAttribute.setClickActionAndroid(true);
21     SingleResourceViewAttribute room = new
22         SingleResourceViewAttribute(roomAttribute);
23     officeAttributes.add(room);
24
25     return officeAttributes;
26 }
```

Im Anhang befinden sich unter Listing 2 die vollständige Definition einer *DetailView* sowie unter Listing 4 die vollständige Definition einer *CardView*.

Eingebende Views

Bei der *InputView* gibt es wieder eine Liste, welche diesmal *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 Ressourcendefinition 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 hinzuzufügen, oder diese zu bearbeiten. Des Weiteren wird eine Subresource nicht in einer *InputView* der Oberresource bearbeitet oder neu angelegt. Dies geschieht in der entsprechenden *View* der Subresource. Die anderen Typen beschränken das *EditText-Widget* auf die angegebenen Typen. So wird beispielsweise bei einem Klick auf ein *PHONE_NUMBER-Feld* die Tastatur im Zahlenmodus ausgefahren.

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 dass entsprechende Feld auszufüllen. Mit der Kombination von *checkPattern* und *errorText* bekommt der Nutzer des Generators die Möglichkeit, 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.

Das Listing 3.8 stellt dar, wie ein *InputViewAttribute* für eine *InputView* definiert werden muss. Das hier initialisierte Attribut ist vom Type *TEXT* und wird mit dem *attributeName* *roomNumber* seinem zugehörigen Attribut der Ressource zugewiesen. Auch wird hier wieder ein Label vergeben, daneben den *Hint-Text* *Room* sowie der *Missing-Text* *Room is missing!*. Im Anhang unter Listing 3 befindet sich eine vollständige Definition einer *InputView*.

Listing 3.8: Definition eines *InputViewAttributes* einer *InputView*.

```

1 InputViewAttribute room = new InputViewAttribute(
2 "roomNumber", ViewAttribute.AttributeType.TEXT,
3 "Room", "Room is missing!");
4 room.setAttributeLabel("Room");
5 inputViewAttributes.add(room);

```

3.3.6 Analyse und Design allgemeiner 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 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 werden um die Farbe für den *Toolbar*-Text erweitert.

Mit dem Wissen, konnte eine Erweiterung des *Enfield-Modells* designed werden, welches in Abbildung 3.16 dargestellt ist.

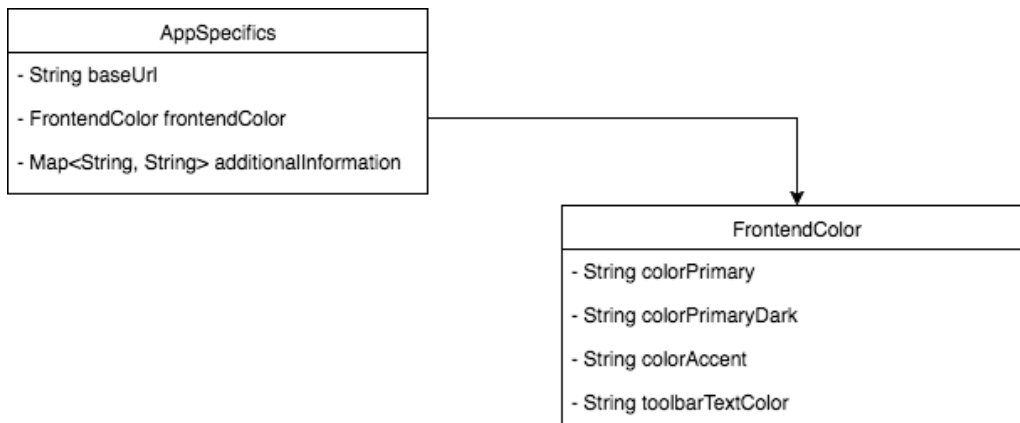


Abbildung 3.16: Aufbau des *AppSpecifics* Objekt zur Erweiterung des *Enfield-Modells*.

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

4 Lösung

In diesem Kapitel wird der in dieser Arbeit entwickelte Software-Generator *Welling* vorgestellt. *Welling* nutzt das zuvor beschriebene *Meta-Modell*, um Android Applikationen generieren zu lassen.

4.1 Design des Software-Generators

In diesem Kapitel soll auf die Problematik eines Software-Generator zu designen eingegangen werden. Ein funktionsfähiger Software-Generator benötigt neben einem geeigneten *Meta-Modell* einen sinnvollen Aufbau. Der Aufbau bestimmt im Zusammenspiel mit dem *Meta-Modell* an welcher Stelle im zeitlichen Verlauf, welche Klassen der Android Applikation generiert werden. Das ist wichtig, da für eine Anwendung die ausschließlich eine Liste darstellen beispielsweise keine Klassen für die Neuanlage von Datensätzen generiert werden sollen. Auch gibt es in Android Applikationen Abhängigkeiten zwischen verschiedenen Klassen, so müssen beispielsweise alle genutzten *Activities* in der sogenannten *AndroidManifest.xml* registriert werden. Oder alle verwendeten Strings, sollten nicht im Programmcode stehen, sondern diese sollten ausgelagert in einer *strings.xml* zu finden sein. Im Programmcode werden diese Strings dann mit Identifikatoren referenziert. Der Generator muss in der Lage sein diese Abhängigkeiten in der Applikation darzustellen.

Eine Android Applikation besteht neben Java- und XML-Klassen zusätzlich noch aus *Gradle*-Dateien und *Java Archiven (JARs)*. Nicht alle Dateien zu generieren macht durchaus Sinn. Bei manchen der Dateien ist es besser, diese an die entsprechende Stelle zu kopieren.

4.1.1 *JavaPoet*

JavaPoet ist ein Java *API*, welches ermöglicht Java-Klassen zu generieren [11]. Hierfür wird die zu generierende Klasse programmiert. Mit Hilfe von nur ein paar Schlüsselwörtern ist es 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, dass gültige Konventionen und Regeln von Java eingehalten werden. So ist der grundsätzlich korrekte Aufbau einer Java-Klasse bereits vorab sichergestellt.

Listing 4.1 zeigt ein einfaches Beispiel zur Generierung einer Hello-World-Klasse und Listing 4.2 zeigt das Ergebnis nach der Ausführung des Beispiels.

Listing 4.1: Beispiel für die Generation einer Hallo-World-Klasse mit JavaPoet [11].

```

1 MethodSpec main = MethodSpec.methodBuilder("main")
2   .addModifiers(Modifier.PUBLIC, Modifier.STATIC)
3   .returns(void.class)
4   .addParameter(String[].class, "args")
5   .addStatement("$T.out.println($S)", System.class, "Hello,
   JavaPoet!")
6   .build();
7
8 TypeSpec helloWorld = TypeSpec.classBuilder("HelloWorld")
9   .addModifiers(Modifier.PUBLIC, Modifier.FINAL)
10  .addMethod(main)
11  .build();
12
13 JavaFile javaFile = JavaFile.builder("com.example.helloworld",
   helloWorld)
14  .build();

```

Listing 4.2: Ergebnis der Generation von Listing 4.1 [11].

```

1 package com.example.helloworld;
2
3 public final class HelloWorld {
4     public static void main(String[] args) {
5         System.out.println("Hello, JavaPoet!");
6     }
7 }

```

4.1.2 Generierung anderer Datentypen

Neben Java-Klassen besitzt der Quellcode einer Android Applikation auch *XML*-Dateien und *Gradle*-Dateien. Für diese Typen muss eine andere Möglichkeit der Generierung gewählt werden. Hierfür liefert GGenerierung von Mobilien Applikationen basierend auf

REST Architekturen (GeMARA) mit der Klasse *GeneratedFile* eine Möglichkeit. Diese Klasse besitzt die beiden Methoden *append(String content)* und *appendln(String content)*, welche 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, jede beliebige Textstruktur zu erzeugen. Jedoch liefert diese Klasse keinerlei Validierung, die Datei wird generiert, egal ob die Struktur gültig ist oder nicht.

Listing 4.3 erzeugt eine in Listing 4.4 dargestellte Datei *test.xml* unter dem Verzeichnis *generated*.

Listing 4.3: 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
7 xmlns:android=\"http://schemas.android.com/apk/res/android\"
8 xmlns:app=\"http://schemas.android.com/apk/res-auto\">");
9         appendln("<item android:id=\"@+id/saveItem\"");
10        appendln("android:title=\"@string/save\"");
11        appendln("app:showAsAction=\"always\">");
12        appendln("<\\menu>");
13    }
14
15    @Override
16    protected String getFileName() {
17        return "test.xml";
18    }
19
20    @Override
21    protected String getDirectoryName() {
22        return "/generated";
23    }
24 }
```


Listing 4.4: Erzeugte XML-Datei durch den Quellcode von Listing 4.3.

```

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>

```

4.1.3 Ablauf der Generierung

Um eine Android Applikation generieren zu lassen, müssen nicht alle Klassen generiert werden. Es können auch Überlegungen angestrebt werden, generische Klassen einfach im Generator abzulegen und bei Bedarf zu kopieren. Diese Methode wurde verworfen, da 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 der *Import*-Anweisungen. Eine weitere Überlegung wäre, 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.

4.1.4 Aufbau des Generators

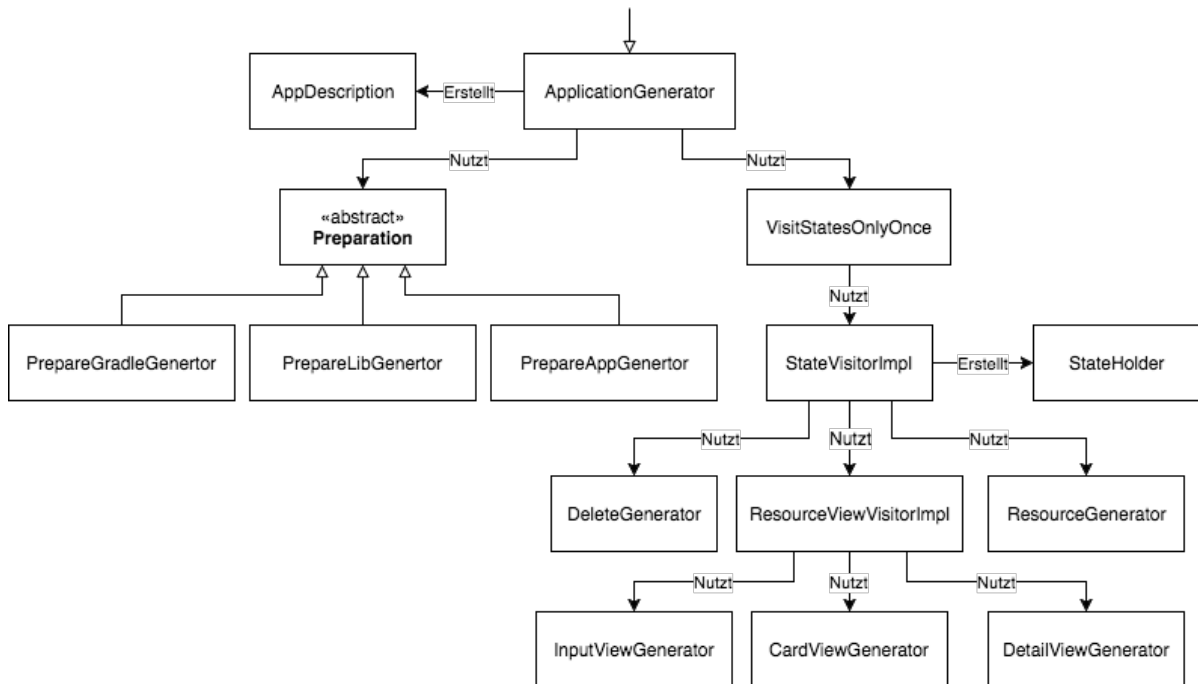
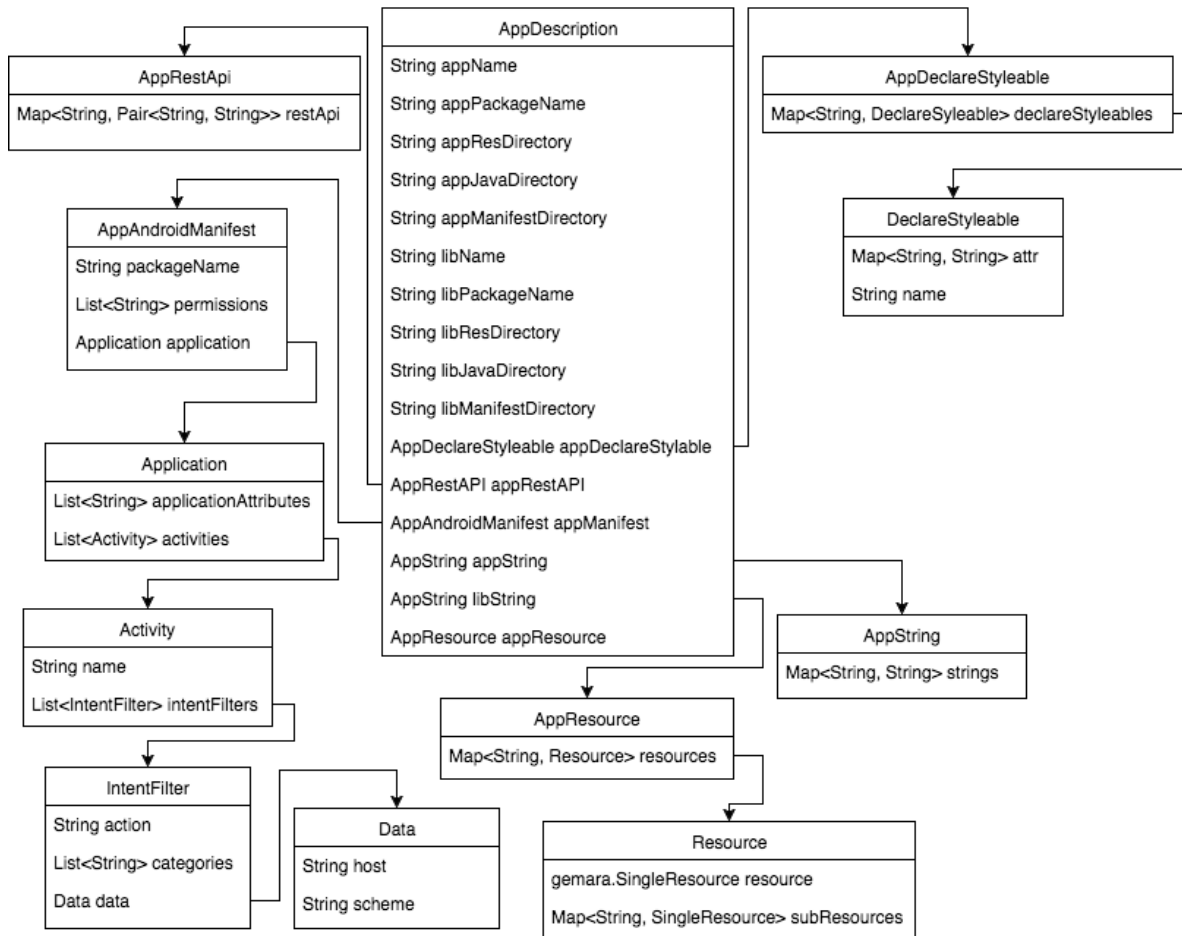


Abbildung 4.1: Aufbau des Android-Generators Welling.

Die Klasse *ApplicationGenerator* ist der Einstiegspunkt des Projekts. Sie erwartet im Konstruktor ein *Enfield-Modell* Objekt. Wie der Abbildung 4.1 entnommen werden kann, lässt sich das Projekt in drei Teilbereiche gliedern. Der erste Bereich erzeugt ein *App-Description* Objekt (Abbildung 4.2), 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-Modell* 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 4.2: Aufbau des *AppDescription* Objekts.

In der *AppDescription* werden alle allgemeinen Daten durch den Generator gereicht, welche an vielen Stellen benötigt werden. Zum Beispiel der Name der Anwendung oder der Bibliothek. In jeder Java-Klasse wird der Paketname 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 Informationen können einfach aus dem *Enfield-Modell* abgelesen werden. Auch die Ressourcen und die jeweiligen Subressourcen können direkt aus dem *Meta-Modell* entnommen werden. Dies ist der Teil des Initialisierens der *AppDescription*. Alle bereits 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 in der Bibliothek. Diese Dateien enthalten neben dem Applikationsnamen, beziehungsweise des Bibliotheksnamens auch viele Strings, die erst beispielsweise in einem Fragment auftauchen. Jedoch müssen die benötigten Datensätze in der *strings.xml* eingetragen werden. Um zu verhindern, dass der Generator die *strings.xml* mehrfach erweitern muss, wird die *AppDescription* in den entsprechenden *AppString*-Attributen *appString* beziehungsweise *libString* erweitert.

Auch das *AndroidManifest* wächst mit der Anwendung. So muss jede benutzte *Activity* dort eingetragen sein, andernfalls kann diese nicht genutzt werden. Am Anfang des Generierens ist die genaue Anzahl und die Namen der *Activities* unbekannt, weswegen der Generator diese beim Erzeugen zur *AppDescription* hinzufügen muss. Das Attribut *appDeclareStyleable* enthält alle *CustomViews*, 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 *Relationstypen* zu welchen Endpunkten gehören. Anfangs sind diese jedoch ebenfalls 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. 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 deshalb nicht bei jeder Ergänzung einer der beschriebenen Dateien diese Aufrufen, den neuen Datensatz aufwändig hinzufügen und die Datei wieder abspeichern. Stattdessen muss der Generator die Datei nur einmal schreiben, da er zu Beginn des Schreibvorgangs 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 drei 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 Projektebene eine *settings.gradle* erzeugt oder in der Applikation sowie der Bibliothek jeweils eine *build.gradle*.

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

Der Bereich, welcher die Bibliothek initialisiert, ist der Größte. Er generiert alle generell benötigten Klassen. Darunter fallen die Klassen der Netzwerkkommunikation, die Klasse für das Link-Objekt sowie das Interface *Resource*. Des Weiteren werden auch die größten Teile der in der Abbildung 3.8 abgebildeten generischen Klassen erzeugt. Auch werden die *CustomViews*, welche definitiv benötigt werden, bereits erzeugt. Dazu gehören auch die benötigten *XML*-Dateien. Für die Bibliothek kann beispielsweise das Manifest bereits erzeugt werden, 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 aufzubereiten. 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, dass zu jedem *State* alle benötigten Klassen und Dateien generiert werden.

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

Gelangt der Generator zu einem *State*, wird mit dem *IStateVisitor* identifiziert von welchem Typ dieser ist. Mögliche *Statetypen* sind: ein *State*, welcher einen *GET-Request* auf eine einzelne Ressource oder auf eine Collection beschreibt oder *States*, welche einen *POST*-, *PUT*- oder *DELETE-Request* repräsentieren. 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. Diese Redundanz garantiert, dass auf jeden Fall eine Ressource zum betreffenden *State* existiert.

Neben diesen Ressource-Klassen, wird auch ein *StateHolder*-Objekt erstellt. Die Abbildung 4.3 repräsentiert dieses.

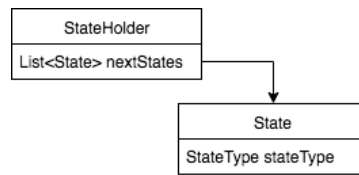


Abbildung 4.3: Aufbau des *StateHolder* Objekts.

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-Modell*, 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 wird lediglich ein *DialogFragment* erzeugt, welches für das Löschen verwendet wird.

Für die anderen *States* werden mehr Klassen und Dateien benötigt. Außerdem werden die *ResourceViews* (Kapitel 3.3.5) 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 ResourceViewVisitor* 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.

4 Lösung

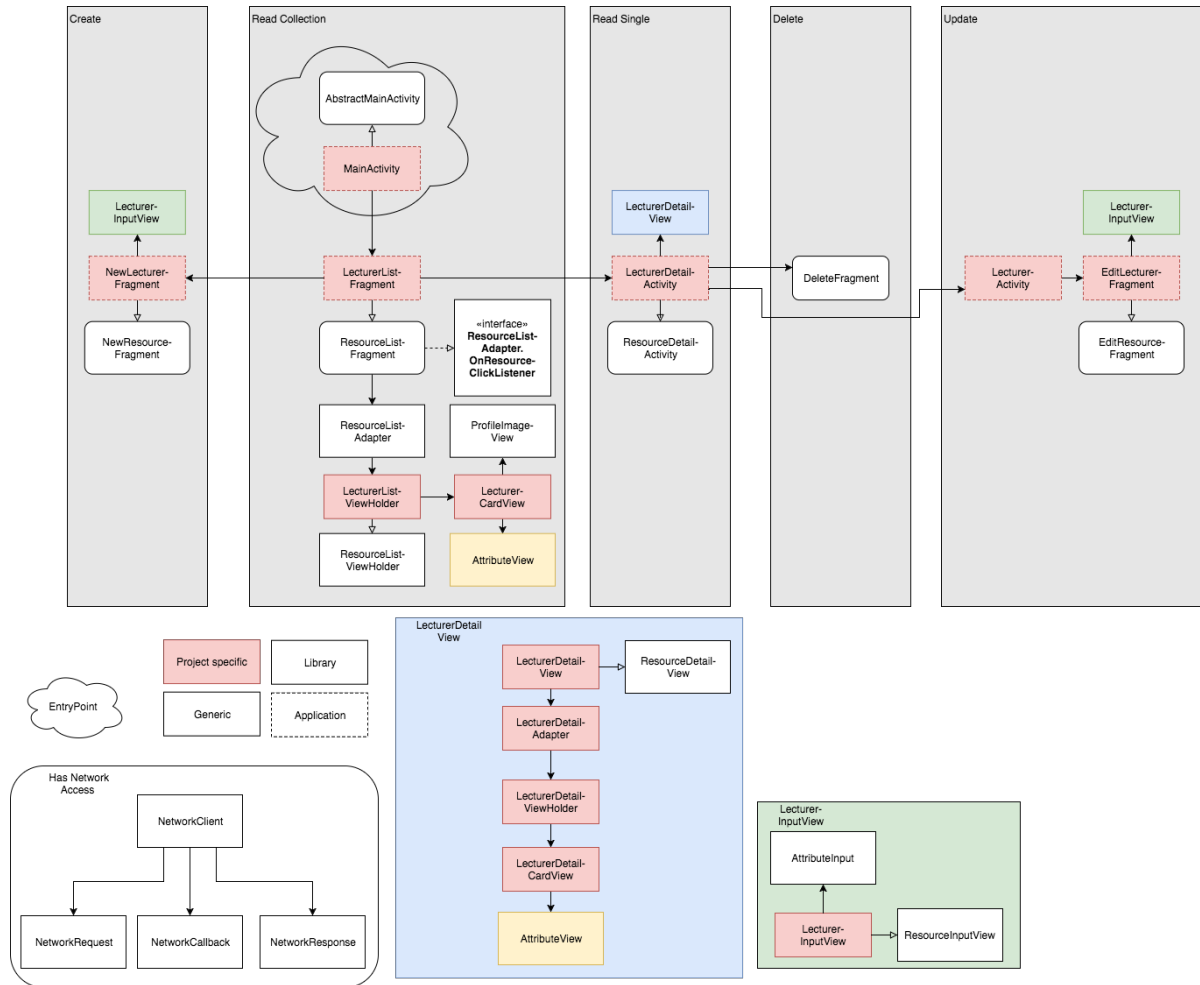


Abbildung 4.4: Aufbau der Dozenten Applikation mit Einteilung in spezifische *States*.

In Abbildung 4.4 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. Dabei werden in einem der Generatoren nicht nur die Java-Klassen für Applikation oder Bibliothek, sondern auch alle benötigten *XML*-Dateien erzeugt.

So ist beispielsweise der *DetailViewGenerator* dafür verantwortlich, dass auf Seite der Applikation, die *LecturerDetailActivity* inklusive ihrer XML-Datei erzeugt wird. Er muss weitergehend auch diese *Activity* in die *AppDescription* im Bereich des *Manifests* hinterlegen. Im Bereich der Bibliothek muss dafür gesorgt werden, dass die generischen Klassen *ResourceDetailActivity*, *ResourceDetailView* sowie die spezifischen Klassen: *LecturerDetailView*, *LecturerDetailAdapter*, *LecturerDetailViewHolder*, *LecturerDetailCardView* 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. Ob eine *Activity* mit einer *CollapsingToolbar* verwendet wird oder ob ein einfaches *Fragment* zur Detailanzeige ausreichend ist, hängt davon ab, ob die Ressource ein Bild besitzt oder nicht. 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.

4.2 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 Integrierte Entwicklungsumgebung (IDE) beispielsweise 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 4.5: *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 4.5 zeigt das *Makefile*, welches die Möglichkeit bietet, 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 anhand einer Beispielanwendung

Dieses Kapitel befasst sich mit Pro und Contra des Generierens einer Android Applikation, nach der in dieser Arbeit vorgestellten Methode. Hierfür wird die Erstellung und die Benutzung des *Meta-Modells* genauer beschrieben, dabei werden die Vorteile und Nachteile des Modells dargestellt. Anschließend wird die Komplexität der Anwendung genauer betrachtet und die Zeitaufwände für die Entwicklung und Wartung des Generators erörtert.

5.1 Erstellung und Nutzung des *Meta-Modells*

Im Vergleich zum Umfang der Entwicklung einer kompletten Anwendung, reduziert die Nutzung des Generators den Aufwand erheblich. Die im Anhang befindlichen Listings 2, 3 und 4 zeigen den kompletten Aufwand der Beschreibung der Anwendung. Auch wenn diese nur ein Teil der benötigten Informationen sind, kann der Rest vernachlässigt werden. Der zusätzlich benötigte Teil ist die Kernbeschreibung der Generierung des *Backends*. So ist dieser semantisch stark diesem Bereich zugeordnet und dort essentiell. Durch diesen Umstand werden diese Informationen als gegeben betrachtet.

Die Fehleranfälligkeit bei der Nutzung des *Meta-Modells* im Gegensatz zur Entwicklung einer kompletten Anwendung ist wesentlich geringer. Die Erzeugung des *Meta-Modells* ist sehr viel eingeschränkter in seinen Möglichkeiten, wodurch die Möglichkeit, Fehler zu machen bedeutend reduziert wird. Das *Meta-Modell* liefert in gewisser Weise einen Plan, wie etwas beschrieben werden muss. Bei der Eigenentwicklung einer Anwendung ist der Entwickler viel freier in der gesamten Handhabung, was das Fehlerpotenzial erhöht.

Jedoch bringt diese Einschränkung durchaus auch Nachteile mit sich. Im Moment ist es beispielsweise nur möglich einer Ressource ein oder kein Bild zuzuweisen. Auch kann der Nutzer lediglich bestimmen, ob dieses Bild in der Karte einer Ressource, in der Liste mit allen Ressourcen dieser Art, auf der linken oder rechten Seite angezeigt werden soll. Für die Detail-Ansicht hat der Nutzer des Generators keine Möglichkeit zu bestimmen, wie das Bild angezeigt werden soll. Auch bleibt zur Anzeige der Informationen einer

Ressource lediglich die Möglichkeit diese in Listenform darzustellen. Er kann nur die Reihenfolge und eine mögliche Gruppierung bestimmen und in der Detail-Ansicht müssen diese Informationen zusätzlich in Kategorien gruppiert sein.

In der aktuellen Version kann der Benutzer des Generators keine eigene Funktionen mit dem Klick auf ein Attribut ausführen, sondern ausschließlich ein Subset von vordefinierten Funktionen. Das Gleiche gilt auch für die Icons, welche in der Karte vor den einzelnen Attributen sichtbar sind. Es gibt im Moment keine Möglichkeit dort eigene Icons anzeigen zu lassen.

5.2 Zeitaufwände und Komplexität

Der gesamte Generator ist in seiner Entwicklung sehr zeitaufwändig. Durch das Analysieren der Anforderungen und das Entwickeln einer *domänenspezifische Sprache (DSL)*, ist dieser Zeitaufwand nur dann gerechtfertigt, wenn mithilfe des Generators viele Anwendungen generiert werden können. Die Neuentwicklung einer Android-Applikation mit dem zuvor beschriebenen Funktionsumfang bedarf einen ungefähren Zeitaufwand von ca. drei Arbeitstagen. Wobei der Zeitaufwand für den Generator ca. zwei bis zweieinhalb Monate beträgt.

Der Aufwand für die Wartung des Generators ist auch ziemlich hoch. Da Android sich ständig weiterentwickelt und eine Umstellung auf das *OpenJDK* erfolgen soll [9], ist anzunehmen, dass sich in Zukunft auch die Art und Weise der Android Programmierung ändern wird. Sollte dies der Fall sein, müssten im kompletten Generator Anpassungen gemacht werden. Diese sind sehr zeitaufwändig, da der Generator, wie Abbildung 4.1 verdeutlicht, sehr komplex ist.

Auch ist die Komplexität der erzeugten Applikation sehr hoch. Die Komplexität des Generators zu reduzieren, wurde mit einer höheren Komplexität der Applikation bezahlt. Diese Komplexität rührt daher, dass zur Einteilung in spezifische und generische Codebereiche, der vorhandene Programmcode so weit wie möglich abstrahiert wurde. Diese Abstraktion führt dazu, dass sich die Anzahl der benötigten Klassen mindestens verdoppelt, da davon ausgehen kann, dass zu jeder spezifischen Klasse mindestens eine generische Klasse erzeugt werden muss. Es können zwar einige abstrakten Klassen von mehreren spezifischen Klassen benutzt werden, jedoch ist in diesem Beispiel diese Wiederverwendung vernachlässigbar. Da die Anzahl der mehrfach benutzen abstrakten Klassen gegenüber dem direkten Vergleich von spezifischen zu generischen Klassen kaum ins Gewicht fällt. Mit der Anzahl der Klassen, haben sich auch die Abhängigkeiten innerhalb der Klassen erhöht. Dadurch ist beispielsweise die Fehleranalyse vor allem während der Entwicklung sehr aufwändig. Auch das Vorgehen, dass nicht eine Anwendung im üblichen Sinn erzeugt wird, sondern das Komponenten in einer Bibliothek erzeugt werden,

steigert den Umfang der Applikation. Bei hardware-schwächeren Endgeräten, könnte dieser Umstand zu Problemen mit der Performance führen. Diese Performanceprobleme entstehen durch die größere Verschachtelung einzelner *View*-Klassen. Die *View*-Klasse wird oft ohne das Bewusstsein um deren Komplexität verwendet. Diese Klasse hat die Aufgabe den anzuzeigenden Inhalt soweit aufzubereiten um ihn auf dem Display anzuzeigen. Wird diese Klasse verschachtelt, so wird der Rechenaufwand für das Endgerät bedeutend erhöht. Aus diesem Grund sind flache *View*-Strukturen vorzuziehen, weshalb das Endgerät mehr Rechenleistung benötigt, um die Anwendung ohne Ruckler darzustellen.

Gegen den großen zeitlichen Aufwand spricht die Einsparung von Zeit beim Generieren neuer Applikationen. Die Beschreibung eines *Meta-Modells* mit allen benötigten Angaben und Informationen, welches die Erzeugung einer Backend-Anwendung inkludiert, benötigt nur noch ungefähr eine Stunde.

6 Zusammenfassung

Im diesem Kapitel wird die gesamte Ausarbeitung noch einmal zusammengefasst. Dabei spiegelt diese Zusammenfassung auch noch einmal den Aufbau der Arbeit dar. Abschließend werden mögliche Ausblicke vorgestellt. Diese beinhalten Erweiterungen, um die der Software-Generator erweitert werden könnte.

6.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde ein Generator für Android Applikationen als Teilprojekt des Generators GEneration von Mobilen Applikationen basierend auf REST Architekturen (GeMARA) entwickelt. Ein Ziel dabei war, dem Leser die grundsätzliche Problematik bei der Entwicklung von Software-Generatoren näher zu bringen. Es wurde erklärt weswegen ein Generator ein *Meta-Modell* benötigt und mögliche Modelle vorgestellt.

Bei der Vorstellung der *Meta-Modelle* wurde aufgezeigt, welche Vor- und Nachteile das jeweilige Modell besitzt. So wurde bei dem Android-spezifischen Modell gezeigt, dass dieses flexibler im Bereich der Funktionalitäten und des Ablaufes innerhalb der Anwendung ist. Jedoch ist es nicht oder nur schwer möglich dieses Modell für einen anderen *Client* mitzuverwenden. Bei der Vorstellung des universellen Modells wurden die bei der Analyse angewendeten Fragen aufgezeigt. Die Verdeutlichen sollen, was bei der Entwicklung von *Meta-Modellen* alles berücksichtigt werden muss.

Es wurde darauf eingegangen, dass das gegebene *Enfield-Meta-Modell* nicht an einer Stelle, sondern an den entsprechenden Stellen in den *States* erweitert wird. Dies hat zum Vorteil, dass der Generator durch das Iterieren über die *States* mit Hilfe der Transitionen, eine Art Fahrplan der Applikation besitzt und zur passenden Stelle alle relevanten Informationen zur Verfügung hat.

Anhand von Codebeispielen wurde gezeigt, wie die *Views* modelliert werden müssen und wie das Ergebnis aussieht. Besonders wurde darauf eingegangen, welche Möglichkeiten der Benutzer des Generators besitzt, um die *Views* zu gestalten. Zu den Gestaltungsmöglichkeiten der Oberfläche wurde außerdem aufgezeigt, welche möglichen Interaktionen bei einem Klick ausgelöst werden können.

Auch wurde dem Leser näher gebracht, wie der Generator für eine Android Applikation funktioniert. Es wurde das Java *Application Programming Interface (API) JavaPoet* vorgestellt, mit dessen Hilfe die Java-Klassen erzeugt werden können. Daneben wurde auch aufgezeigt, wie die anderen Dateien erzeugt werden können. Neben dem reinen Erzeugen wurde der Ablauf im Generator vorgestellt. Es wurde gezeigt, dass sich dieser in drei Bereiche gliedert. Jeder dieser Bereiche wurde vorgestellt und auf seine Besonderheiten hingewiesen. Dadurch sollte ein Verständnis über die Funktionsweise vermittelt werden.

6.2 Ausblick

Im letzten Kapitel der Ausarbeitung sollen Ideen und mögliche Erweiterungen des Meta-Modells sowie des Software-Generators für Android Applikationen vorgestellt werden.

In der ersten Version des Generators ist bisher nur möglich, eine Ressource als *Primärressource* zu definieren. Jedoch würde es die Möglichkeit geben, mehrere Ressourcen zu definieren und in der Applikation mit Hilfe eines *Navigation-Drawers* zwischen diesen umzuschalten. Der Grundstein dafür ist bereits in dieser Version gelegt worden. Neben den drei, in dieser Arbeit beschriebenen, *ResourceViews* wurde bereits eine vierte *View* im *Meta-Modell* eingefügt. Die *NavigationDrawerResourceView* mit deren Hilfe der *Drawer* in der Applikation beschrieben werden könnte.

Außerdem sind im Moment die *Views* auf ein Bild beschränkt, in einer späteren Version, könnte diese Begrenzung aufgehoben werden und dadurch einer Ressource mehrere Bilder als Attribute zugeteilt werden. Dafür müsste jedoch auch das Modell dahingehend erweitert werden, das der Generator weiß, welches Bild als Titelbild verwendet wird. Dieses würde dann weiterhin in der *CollapsingToolbar* der Detailansicht angezeigt werden. Da die *CollapsingToolbar* ein Style-Element von Material Design ist, sollte dieses so beibehalten werden. Jedoch müsste überlegt werden, wie die zusätzlichen Bilder angezeigt werden sollen.

Auch wurde in der Ausarbeitung darauf eingegangen, dass einem Attribut in einer *InputResourceView* ein *checkPattern* sowie ein *errorText* mitgegeben werden kann. Jedoch werden aktuell diese Eigenschaften nicht zur Validierung der Eingabe herangezogen. Zusätzlich könnte für die Checks noch angegeben werden, ob es optionale Eingabefelder gibt. Im Moment müssen alle angegebenen Felder befüllt werden.

Da es für Android-Anwendungen eher unüblich ist Bilder zu einer Ressource, durch das Hochladen dieses, hinzuzufügen, wurde im ersten Entwurf auf das Feature verzichtet. In Zukunft wäre es jedoch denkbar, diese Möglichkeit zu unterstützen. Eine weitere nützliche Erweiterung wäre die Suche nach einer bestimmten Ressource. Dieses Feature war zwar Anfangs bereits angedacht, wurde jedoch erst einmal wegen einer geringeren Priorität hinten angestellt.

Abbildungsverzeichnis

1.1	Prognose zur Anzahl der Smartphone-Nutzer weltweit von 2012 bis 2020 (in Milliarden) [17].	1
1.2	Der weltweite Marktanteil von Smartphone-Betriebssysteme [2].	2
2.1	Aufbau eines REpresentational State Transfer-Application Programming Interface mit Hilfe eines <i>endlichen Automaten</i>	7
2.2	Aufbau von GeMARA	13
3.1	Darstellung des <i>API</i> der Beispielanwendung.	18
3.2	<i>RecyclerView</i> zur Darstellung aller Dozenten.	19
3.3	Ausschnitt der <i>View</i> zur Erstellung eines Dozenten.	20
3.4	Fehlermeldung bei der Neuanlage eines Dozenten.	20
3.5	Detailansicht eines Dozenten.	21
3.6	<i>Dialog</i> zum Löschen eines Dozenten.	21
3.7	<i>DateTimePicker-Widget</i> zur Datumsauswahl.	22
3.8	Aufbau der Referenzimplementierung.	24
3.9	Vereinfachter Aufbau des <i>Enfield-Meta-Modells</i>	27
3.10	Möglicher Aufbau eines <i>Android-Meta-Modells</i>	27
3.11	Vereinfachter Aufbau des erweiterten <i>Enfield-Meta-Modells</i>	29
3.12	Beispiel einer <i>CardView</i> eines Dozenten nach Material Design.	30
3.13	Beispiel einer <i>DetailView</i> eines Dozenten nach Material Design.	32
3.14	Beispiel einer <i>View</i> zum Anlegen eines Dozenten.	33
3.15	Aufbau der <i>Views</i> zur Erweiterung des <i>Enfield-Modells</i>	35
3.16	Aufbau des <i>AppSpecifics</i> Objekt zur Erweiterung des <i>Enfield-Modells</i> . . .	40
4.1	Aufbau des Android-Generators Welling.	45
4.2	Aufbau des <i>AppDescription</i> Objekts.	46
4.3	Aufbau des <i>StateHolder</i> Objekts.	49
4.4	Aufbau der Dozenten Applikation mit Einteilung in spezifische <i>States</i> . . .	50

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] *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/>.
- [5] *CollapsingToolbarLayout*. Eingesehen am 28.02.17.
URL: <https://developer.android.com/reference/android/support/design/widget/CollapsingToolbarLayout.html>.
- [6] *CRUD Admin Generator*. Eingesehen am 17.11.16.
URL: <http://crud-admin-generator.com/>.
- [7] Roy Thomas Fielding.
„Architectural Styles and the Design of Network-based Software Architectures“. dissertation. University of California, Irvine, 2000.
URL: <http://docserver.bis.uni-oldenburg.de/publikationen/dissertation/2002/applou02/applou02.html>.
- [8] Martin Fowler. *Domain Specific Languages*. 1st. Addison-Wesley Professional, 2010. ISBN: 0321712943, 9780321712943.
- [9] *Google confirms next Android version will use Oracle's open-source OpenJDK for Java APIs*. Eingesehen am 05.03.17.
URL: <http://venturebeat.com/2015/12/29/google-confirms-next-android-version-wont-use-oracles-proprietary-java-apis/>.
- [10] Paul Hudak. „Domain-specific languages“. In: *Handbook of Programming Languages* 3.39-60 (1997), S. 21.

- [11] *JavaPoet*. Eingesehen am 02.03.17.
URL: <https://github.com/square/javapoet>.
- [12] *Material Design*. Eingesehen am 28.02.17.
URL: <https://material.io/guidelines/>.
- [13] *open handset alliance*. Eingesehen am 06.01.17.
URL: <http://www.openhandsetalliance.com/index.html>.
- [14] *REST APIs must be hypertext-driven*. Eingesehen am 27.03.17. URL: <http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>.
- [15] Thomas Stahl, Markus Voelter und Krzysztof Czarnecki.
Model-Driven Software Development: Technology, Engineering, Management.
John Wiley & Sons, 2006. ISBN: 0470025700.
- [16] *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/>.
- [17] *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 27. März 2017

Anhang

Im Anhang befinden sich Codebeispiele und Ergänzungen zu den vorher angesprochenen Themen: *Enfield-Modell* der Referenzimplementierung; Erstellung einer *DetailView*; Erstellung einer *InputView* sowie die Erstellung einer *CardView*. Auch diese Listings sind teilweise auf das wesentliche reduziert, da sie lediglich ein besseres Verständnis über die jeweiligen Themenbereiche vermitteln sollen.

Listing 1: Beschreibung des *Enfield-Modell* der Referenzimplementierung.

```

1 class MyEnfieldModel {
2
3     private final Model metaModel;
4     private SingleResource lecturerResource;
5     private SingleResource chargeResource;
6     private GetDispatcherState dispatcherState;
7     private GetPrimarySingleResourceByIdState
8         getLecturerByIdState;
9     private GetPrimaryCollectionResourceByQueryState
10        getCollectionOfLecturersState;
11    private GetPrimarySingleResourceByIdState
12        getChargeByIdState;
13    private GetPrimaryCollectionResourceByQueryState
14        getCollectionOfChargesState;
15
16    public MyEnfieldModel() {
17        this.metaModel = new Model();
18
19        this.metaModel.setProducerName("fhws");
20        this.metaModel.setPackagePrefix("de.fhws.applab.gemara");
21        this.metaModel.setProjectName("Lecturer");
22
23        this.metaModel.setAppSpecifics(getAppSpecifics());
24    }
25
26    public Model create() {
27
28        createSingleResourceLecturer();
29        createDispatcherState();
30        createGetLecturerByIdState();
31        createGetCollectionOfLecturersState();
32        createDeleteLecturerState();
33        createPostNewLecturerState();
34        createUpdateLecturerState();
35        createPostLecturerImageState();
36        createGetLecturerImageState();
37        createGetCollectionOfChargesState();
38        createGetChargeByIdState();
39        createPostNewChargeState();
40        createUpdateChargeState();
41        createDeleteChargeState();
42

```

```

43     return this.metaModel;
44 }
45
46 private AppSpecifcs getAppSpecifcs() {
47     AppSpecifcs appSpecifcs = new
48         AppSpecifcs("https://apistaging.fiw.fhws.de/mig/api/");
49     appSpecifcs.setFrontendColor(getFrontendColor());
50
51     return appSpecifcs;
52 }
53
54 private FrontendColor getFrontendColor() {
55     try {
56         return new FrontendColor(
57             "#3F51B5", "#303F9F", "#FF4081", "#fff");
58     } catch (InputException ex) {
59         return null;
60     }
61 }
62
63 private void createSingleResourceLecturer() {
64     this.metaModel.addSingleResource("Lecturer");
65     this.lecturerResource =
66         this.metaModel.getSingleResource("Lecturer");
67     this.lecturerResource.setModel(this.metaModel);
68     this.lecturerResource.setResourceName("Lecturer");
69     this.lecturerResource.setMediaType(
70         "application/vnd.fhws-lecturer.default+json");
71     final SimpleAttribute title = new SimpleAttribute("title",
72         SimpleDatatype.STRING);
73     final SimpleAttribute firstName =
74         new SimpleAttribute("firstName", SimpleDatatype.STRING);
75     final SimpleAttribute lastName =
76         new SimpleAttribute("lastName", SimpleDatatype.STRING);
77     final SimpleAttribute email =
78         new SimpleAttribute("email", SimpleDatatype.STRING);
79     final SimpleAttribute phone =
80         new SimpleAttribute("phone", SimpleDatatype.STRING);
81     final SimpleAttribute address =
82         new SimpleAttribute("address", SimpleDatatype.STRING);
83     final SimpleAttribute roomNumber =
84         new SimpleAttribute("roomNumber", SimpleDatatype.STRING);
85     final SimpleAttribute homepage =
86         new SimpleAttribute("homepage", SimpleDatatype.LINK);

```

```

87
88     createSingleResourceCharge();
89     final ResourceCollectionAttribute charge =
90         new ResourceCollectionAttribute("chargeUrl",
91             this.chargeResource);
92
93     this.lecturerResource.addAttribute(title);
94     this.lecturerResource.addAttribute(firstName);
95     this.lecturerResource.addAttribute(lastName);
96     this.lecturerResource.addAttribute(email);
97     this.lecturerResource.addAttribute(phone);
98     this.lecturerResource.addAttribute(address);
99     this.lecturerResource.addAttribute(roomNumber);
100    this.lecturerResource.addAttribute(homepage);
101    this.lecturerResource.addAttribute(charge);
102
103    addImageAttributeForLecturerResource();
104
105    }
106
107    private void addImageAttributeForLecturerResource() {
108
109        final SimpleAttribute profileImage =
110            new SimpleAttribute("profileImageUrl",
111                SimpleDatatype.IMAGE);
112        profileImage.setBelongsToResource(this.lecturerResource);
113        this.lecturerResource.addAttribute(profileImage);
114        this.lecturerResource.setCaching(new CachingByEtag());
115    }
116
117    private void createSingleResourceCharge() {
118
119        this.metaModel.addSingleResource("Charge");
120
121        this.chargeResource =
122            this.metaModel.getSingleResource("Charge");
123        this.chargeResource.setModel(this.metaModel);
124        this.chargeResource.setResourceName("Charge");
125        this.chargeResource.setMediaType(
126            "application/vnd.fhws-charge.default+json");
127
128        final SimpleAttribute id =
129            new SimpleAttribute("id", SimpleDatatype.INT);
130        final SimpleAttribute titleOfCharge =

```

```

131         new SimpleAttribute("title", SimpleDatatype.STRING);
132     final SimpleAttribute fromDate =
133         new SimpleAttribute("fromDate", SimpleDatatype.DATE);
134     final SimpleAttribute toDate =
135         new SimpleAttribute("toDate", SimpleDatatype.DATE);
136
137     this.chargeResource.addAttribute(id);
138     this.chargeResource.addAttribute(titleOfCharge);
139     this.chargeResource.addAttribute(fromDate);
140     this.chargeResource.addAttribute(toDate);
141 }
142
143 private void createDispatcherState() {
144
145     final GetDispatcherState dispatcherState =
146         new GetDispatcherState();
147     dispatcherState.setName("Dispatcher");
148     dispatcherState.setModel(this.metaModel);
149     this.metaModel.setDispatcherState(dispatcherState);
150     this.dispatcherState = dispatcherState;
151 }
152
153 private void createGetLecturerByIdState() {
154
155     final GetPrimarySingleResourceByIdState getLecturerByIdState =
156         new GetPrimarySingleResourceByIdState();
157     getLecturerByIdState.setName("GetOneLecturer");
158     getLecturerByIdState.setResourceType(this.lecturerResource);
159     getLecturerByIdState.setModel(this.metaModel);
160     this.lecturerResource
161         .setDefaultStateForSelfUri(getLecturerByIdState);
162
163     this.metaModel.addState(getLecturerByIdState.getName(),
164         getLecturerByIdState);
165
166     this.getLecturerByIdState = getLecturerByIdState;
167
168     addLecturerDetailView();
169 }
170
171 private void addLecturerDetailView() {
172
173     final DetailView lecturerDetailView =
174         DetailViewModelGenerator.lecturer();

```



```

175
176     final SingleResourceView resourceView =
177         new SingleResourceView();
178
179     resourceView.setDetailView(lecturerDetailView);
180
181     this.getLecturerByIdState.setSingleResourceView(resourceView);
182 }
183
184 private void createGetCollectionOfLecturersState() {
185
186     final GetPrimaryCollectionResourceByQueryState
187         getAllLecturersCollectionState =
188         new GetPrimaryCollectionResourceByQueryState();
189     getAllLecturersCollectionState.setName("GetAllLecturers");
190     getAllLecturersCollectionState.setModel(this.metaModel);
191     getAllLecturersCollectionState.setResourceType(
192         this.lecturerResource);
193
194     this.dispatcherState.addTransition(
195         new ActionTransition(getAllLecturersCollectionState,
196             "getAllLecturers"));
197     getAllLecturersCollectionState.addTransition(
198         this.getLecturerByIdState);
199
200     this.metaModel.addState(
201         getAllLecturersCollectionState.getName(),
202         getAllLecturersCollectionState);
203
204     this.getCollectionOfLecturersState =
205         getAllLecturersCollectionState;
206
207     addLecturerCardView();
208 }
209
210 private void addLecturerCardView() {
211
212     final CardView lecturerCardView =
213         CardViewModelGenerator.lecturer();
214
215     final SingleResourceView resourceView =
216         new SingleResourceView();
217
218     resourceView.setCardView(lecturerCardView);

```

```

219         this.getCollectionOfLecturersState.setSingleResourceView(
220             resourceView);
221     }
222
223     private void createDeleteLecturerState() {
224
225         final DeletePrimaryResourceState deleteLecturerState =
226             new DeletePrimaryResourceState();
227         deleteLecturerState.setName("DeleteOneLecturer");
228         deleteLecturerState.setResourceType(this.lecturerResource);
229         deleteLecturerState.setModel(this.metaModel);
230
231         deleteLecturerState.addTransition(this.getCollectionOfLecturersState,
232             "getAllLecturers");
233         this.getLecturerByIdState.addTransition(
234             deleteLecturerState, "deleteLecturer");
235
236         this.metaModel.addState(
237             deleteLecturerState.getName(), deleteLecturerState);
238     }
239
240     private void createPostNewLecturerState() {
241
242         final PostPrimaryResourceState postNewLecturerState =
243             new PostPrimaryResourceState();
244         postNewLecturerState.setName("CreateOneLecturer");
245         postNewLecturerState.setResourceType(this.lecturerResource);
246         postNewLecturerState.setModel(this.metaModel);
247         this.getCollectionOfLecturersState.addTransition(
248             new ActionTransition(postNewLecturerState,
249                 "createNewLecturer"));
250
251         this.metaModel.addState(postNewLecturerState.getName(),
252             postNewLecturerState);
253
254         addLecturerInputView(postNewLecturerState);
255     }
256
257     private void addLecturerInputView(AbstractPrimaryState state) {
258
259         final InputView lecturerInputView =
260             InputViewModelGenerator.lecturer();

```

```

261     final SingleResourceView resourceView =
262         new SingleResourceView();
263
264     resourceView.setInputView(lecturerInputView);
265
266     state.setSingleResourceView(resourceView);
267 }
268
269 private void createUpdateLecturerState() {
270
271     final PutPrimaryResourceState updateLecturerState =
272         new PutPrimaryResourceState();
273     updateLecturerState.setName("UpdateLecturer");
274     updateLecturerState.setResourceType(this.lecturerResource);
275     updateLecturerState.setModel(this.metaModel);
276
277     this.getLecturerByIdState.addTransition(
278         new ActionTransition(updateLecturerState,
279 "updateLecturer"));
280
281     this.metaModel.addState(
282         updateLecturerState.getName(), updateLecturerState);
283
284     addLecturerInputView(updateLecturerState);
285 }
286
287 private void createPostLecturerImageState() {
288
289     final PostImageState postProfileImageState =
290         new PostImageState();
291     postProfileImageState.setName("PostProfileImage");
292     postProfileImageState.setResourceType(this.lecturerResource);
293     postProfileImageState.setModel(this.metaModel);
294     postProfileImageState.setImageAttribute(
295         (SimpleAttribute) this.lecturerResource.getAttributeByName(
296 "profileImageUrl"));
297
298     this.getLecturerByIdState.addTransition(
299         postProfileImageState, "postImage");
300     postProfileImageState.setName("PostProfileImage");
301     postProfileImageState.setResourceType(this.lecturerResource);
302     postProfileImageState.setModel(this.metaModel);
303     postProfileImageState.setImageAttribute(
304         (SimpleAttribute) this.lecturerResource.getAttributeByName(

```

```

304         "profileImageUrl"));
305
306         this.metaModel.addState(
307             postProfileImageState.getName(), postProfileImageState);
308     }
309
310     private void createGetLecturerImageState() {
311
312         final GetImageState getProfileImageState = new GetImageState();
313         getProfileImageState.setName("GetProfileImage");
314         getProfileImageState.setResourceType(this.lecturerResource);
315         getProfileImageState.setModel(this.metaModel);
316         getProfileImageState.setImageAttribute(
317             (SimpleAttribute) this.lecturerResource.getAttributeByName(
318                 "profileImageUrl"));
319
320         this.getLecturerByIdState.addTransition(
321             getProfileImageState, "getProfileImage");
322
323         this.metaModel.addState(
324             getProfileImageState.getName(), getProfileImageState);
325     }
326
327     private void createGetCollectionOfChargesState() {
328
329         final GetPrimaryCollectionResourceByQueryState
330             getCollectionOfChargesState =
331             new GetPrimaryCollectionResourceByQueryState();
332         getCollectionOfChargesState.setName("GetAllChargesOfLecturer");
333         getCollectionOfChargesState.setModel(this.metaModel);
334         getCollectionOfChargesState.setResourceType(
335             this.chargeResource);
336
337         this.getLecturerByIdState.addTransition(
338             new ContentTransition(getCollectionOfChargesState));
339
340         this.metaModel.addState(
341             getCollectionOfChargesState.getName(),
342             getCollectionOfChargesState);
343
344         this.getCollectionOfChargesState = getCollectionOfChargesState;
345
346         addChargeCardView();
347     }

```

```

348
349 private void addChargeCardView() {
350
351     final CardView chargeCardView =
352         CardViewModelGenerator.charges();
353
354     final SingleResourceView resourceView =
355         new SingleResourceView();
356
357     resourceView.setCardView(chargeCardView);
358
359     this.getCollectionOfChargesState.setSingleResourceView(
360         resourceView);
361 }
362
363 private void createGetChargeByIdState() {
364
365     final GetPrimarySingleResourceByIdState getChargeByIdState =
366         new GetPrimarySingleResourceByIdState();
367     getChargeByIdState.setName("GetOneChargeOfLecturer");
368     getChargeByIdState.setResourceType(this.chargeResource);
369     getChargeByIdState.setModel(this.metaModel);
370
371     this.chargeResource.setDefaultStateForSelfUri(getChargeByIdState);
372
373     this.getCollectionOfChargesState.addTransition(
374         getChargeByIdState);
375
376     this.metaModel.addState(
377         getChargeByIdState.getName(), getChargeByIdState);
378
379     this.getChargeByIdState = getChargeByIdState;
380
381     addChargeDetailView();
382 }
383
384 private void addChargeDetailView() {
385
386     final DetailView chargeDetailView =
387         DetailViewModelGenerator.charge();
388
389     final SingleResourceView resourceView =
390         new SingleResourceView();

```

```

391     resourceView.setDetailView(chargeDetailView);
392
393     this.getChargeByIdState.setSingleResourceView(resourceView);
394 }
395
396 private void createPostNewChargeState() {
397
398     final PostPrimaryResourceState createSingleChargePrimaryState =
399         new PostPrimaryResourceState();
400     createSingleChargePrimaryState.setName("CreateOneCharge");
401     createSingleChargePrimaryState.setResourceType(
402         this.chargeResource);
403     createSingleChargePrimaryState.setModel(this.metaModel);
404     this.getCollectionOfChargesState.addTransition(
405         new ActionTransition(
406             createSingleChargePrimaryState,
407             "createChargeOfLecturer"));
408
409     this.metaModel.addState(
410         createSingleChargePrimaryState.getName(),
411         createSingleChargePrimaryState);
412
413     addChargeInputView(createSingleChargePrimaryState);
414 }
415
416 private void addChargeInputView(AbstractPrimaryState state) {
417
418     final InputView chargeInputView =
419         InputViewModelGenerator.charge();
420
421     final SingleResourceView resourceView =
422         new SingleResourceView();
423
424     resourceView.setInputView(chargeInputView);
425
426     state.setSingleResourceView(resourceView);
427 }
428
429 private void createUpdateChargeState() {
430
431     final PutPrimaryResourceState editSingleChargePrimaryState =
432         new PutPrimaryResourceState();
433     editSingleChargePrimaryState.setName(
434         "UpdateOneChargeOfLecturer");

```

```

435     editSingleChargePrimaryState.setResourceType(
436         this.chargeResource);
437     editSingleChargePrimaryState.setModel(this.metaModel);
438
439     this.getChargeByIdState.addTransition(
440         new ActionTransition(
441             editSingleChargePrimaryState, "updateCharge"));
442
443     this.metaModel.addState(
444         editSingleChargePrimaryState.getName(),
445         editSingleChargePrimaryState);
446
447     addChargeInputView(editSingleChargePrimaryState);
448 }
449
450 private void createDeleteChargeState() {
451
452     final DeletePrimaryResourceState deleteSingleChargePrimaryState
453         = new DeletePrimaryResourceState();
454     deleteSingleChargePrimaryState.setName(
455         "DeleteOneChargeOfLecturer");
456     deleteSingleChargePrimaryState.setResourceType(
457         this.chargeResource);
458     deleteSingleChargePrimaryState.setModel(this.metaModel);
459
460     deleteSingleChargePrimaryState.addTransition(
461         this.getCollectionOfChargesState, "getAllCharges");
462     this.getChargeByIdState.addTransition(
463         deleteSingleChargePrimaryState, "deleteCharge");
464
465     this.metaModel.addState(
466         deleteSingleChargePrimaryState.getName(),
467         deleteSingleChargePrimaryState);
468 }
469 }

```

Listing 2: Erstellung einer *DetailView*.

```

1  [...]
2  DetailView detailView;
3
4      try {
5          List<Category> categories = new ArrayList<>();
6
7          DisplayViewAttribute nameAttribute =
8              new DisplayViewAttribute("name",
9                  ViewAttribute.AttributeType.TEXT);
10         GroupResourceViewAttribute name =
11             new GroupResourceViewAttribute(nameAttribute,
12                 getViewTitleAttributes());
13
14         categories.add(new Category("Office",
15             getOfficeResourceViewAttributes()));
16         categories.add(new Category("Contact",
17             getContactResourceViewAttributes()));
18         categories.add(new Category("Charges",
19             getChangeResourceViewAttributes()));
20
21         detailView = new DetailView("Lecturer", name, categories);
22         detailView.setImage(getImage());
23     } catch (DisplayViewException ex) {
24         detailView = null;
25     }
26  [...]
27  private static List<ResourceViewAttribute>
28  getOfficeResourceViewAttributes() {
29
30      List<ResourceViewAttribute> officeAttributes = new ArrayList<>();
31
32      DisplayViewAttribute addressAttribute =
33          new DisplayViewAttribute("address",
34              ViewAttribute.AttributeType.LOCATION);
35      addressAttribute.setAttributeLabel("Address");
36      addressAttribute.setClickActionAndroid(true);
37      SingleResourceViewAttribute address =
38          new SingleResourceViewAttribute(addressAttribute);
39      officeAttributes.add(address);
40
41      DisplayViewAttribute roomAttribute =
42          new DisplayViewAttribute("roomNumber",

```



```
42         ViewAttribute.AttributeType.TEXT);
43     roomAttribute.setAttributeLabel("Room");
44     roomAttribute.setClickActionAndroid(true);
45     SingleResourceViewAttribute room =
46         new SingleResourceViewAttribute(roomAttribute);
47     officeAttributes.add(room);
48
49     return officeAttributes;
50 }
51 [...]
```

Listing 3: Erstellung einer *InputView*.

```
1 [...]
2 List<InputViewAttribute> inputViewAttributes = new ArrayList<>();
3
4 InputViewAttribute title = new InputViewAttribute("title",
5     ViewAttribute.AttributeType.TEXT, "Title", "Title is missing!");
6 title.setAttributeLabel("Title");
7 inputViewAttributes.add(title);
8
9 InputViewAttribute firstName = new InputViewAttribute("firstName",
10     ViewAttribute.AttributeType.TEXT, "Firstname",
11     "Firstname is missing!");
12 firstName.setAttributeLabel("Firstname");
13 inputViewAttributes.add(firstName);
14
15 InputViewAttribute lastName = new InputViewAttribute("lastName",
16     ViewAttribute.AttributeType.TEXT, "Lastname",
17     "LastName is missing!");
18 lastName.setAttributeLabel("Lastname");
19 inputViewAttributes.add(lastName);
20
21 InputViewAttribute mail = new InputViewAttribute("email",
22     ViewAttribute.AttributeType.MAIL, "E-Mail", "E-Mail is missing!");
23 mail.setAttributeLabel("E-Mail");
24 inputViewAttributes.add(mail);
25
26 InputViewAttribute phone = new InputViewAttribute("phone",
27     ViewAttribute.AttributeType.PHONE_NUMBER, "Phone Number",
28     "Phone number is missing!");
29 phone.setAttributeLabel("Phone Number");
30 inputViewAttributes.add(phone);
31
32 InputViewAttribute address = new InputViewAttribute("address",
33     ViewAttribute.AttributeType.TEXT, "Address", "Address is missing!");
34 address.setAttributeLabel("Address");
35 inputViewAttributes.add(address);
36
37 InputViewAttribute room = new InputViewAttribute("roomNumber",
38     ViewAttribute.AttributeType.TEXT, "Room", "Room is missing!");
39 room.setAttributeLabel("Room");
40 inputViewAttributes.add(room);
41
42 InputViewAttribute weLearn = new InputViewAttribute("homepage",
```

```
        ViewAttribute.AttributeType.URL, "welearn",  
36 "welearn URL is missing!");  
37 weLearn.setAttributeLabel("welearn");  
38 inputViewAttributes.add(weLearn);  
39  
40 InputView inputView;  
41 try {  
42     inputView = new InputView("Lecturer", inputViewAttributes);  
43 } catch (InputViewException ex) {  
44     inputView = null;  
45 }  
46 [...]
```

Listing 4: Erstellung einer *CardView*.

```

1  [...]
2  List<ResourceViewAttribute> resourceViewAttributes = new ArrayList<>();
3
4  DisplayViewAttribute titleAttributes = new
    DisplayViewAttribute("title", ViewAttribute.AttributeType.TEXT);
5  titleAttributes.setAttributeLabel("Title");
6  SingleResourceViewAttribute title = new
    SingleResourceViewAttribute(titleAttributes);
7  resourceViewAttributes.add(title);
8
9  DisplayViewAttribute nameAttribute = new DisplayViewAttribute("name",
    ViewAttribute.AttributeType.TEXT);
10 nameAttribute.setFontSize(DisplayViewAttribute.FontSize.LARGE);
11 List<DisplayViewAttribute> nameAttributes = new ArrayList<>();
12
13 DisplayViewAttribute firstNameAttributes = new
    DisplayViewAttribute("firstName", ViewAttribute.AttributeType.TEXT);
14 firstNameAttributes.setAttributeLabel("FirstName");
15 nameAttributes.add(firstNameAttributes);
16
17 DisplayViewAttribute lastNameAttributes = new
    DisplayViewAttribute("lastName", ViewAttribute.AttributeType.TEXT);
18 lastNameAttributes.setAttributeLabel("LastName");
19 nameAttributes.add(lastNameAttributes);
20
21 GroupResourceViewAttribute name;
22 try {
23     nameAttribute.setFontColor("#000");
24     name = new GroupResourceViewAttribute(nameAttribute,
        nameAttributes);
25 } catch (DisplayViewException ex) {
26     name = null;
27 }
28 resourceViewAttributes.add(name);
29
30 DisplayViewAttribute mailAttribute = new DisplayViewAttribute("email",
    ViewAttribute.AttributeType.MAIL);
31 mailAttribute.setAttributeLabel("E-Mail");
32 mailAttribute.setClickActionAndroid(true);
33 SingleResourceViewAttribute mail = new
    SingleResourceViewAttribute(mailAttribute);
34 resourceViewAttributes.add(mail);

```

```

35
36 DisplayViewAttribute phoneAttribute = new DisplayViewAttribute("phone",
    ViewAttribute.AttributeType.PHONE_NUMBER);
37 phoneAttribute.setAttributeLabel("Phone Number");
38 phoneAttribute.setClickActionAndroid(true);
39 SingleResourceViewAttribute phone = new
    SingleResourceViewAttribute(phoneAttribute);
40 resourceViewAttributes.add(phone);
41
42 DisplayViewAttribute addressAttribute = new
    DisplayViewAttribute("address",
    ViewAttribute.AttributeType.LOCATION);
43 addressAttribute.setAttributeLabel("Address");
44 addressAttribute.setClickActionAndroid(true);
45 SingleResourceViewAttribute address = new
    SingleResourceViewAttribute(addressAttribute);
46 resourceViewAttributes.add(address);
47
48 DisplayViewAttribute roomAttribute = new
    DisplayViewAttribute("roomNumber", ViewAttribute.AttributeType.HOME);
49 roomAttribute.setAttributeLabel("Room");
50 roomAttribute.setClickActionAndroid(true);
51 SingleResourceViewAttribute room = new
    SingleResourceViewAttribute(roomAttribute);
52 resourceViewAttributes.add(room);
53
54 DisplayViewAttribute welearnAttribute = new
    DisplayViewAttribute("homepage", ViewAttribute.AttributeType.URL);
55 welearnAttribute.setAttributeLabel("welearn");
56 welearnAttribute.setClickActionAndroid(true);
57 welearnAttribute.setLinkDescription("welearn");
58 SingleResourceViewAttribute welearn = new
    SingleResourceViewAttribute(welearnAttribute);
59 resourceViewAttributes.add(welearn);
60
61 DisplayViewAttribute imageAttribute = new
    DisplayViewAttribute("profileImageUrl",
    ViewAttribute.AttributeType.PICTURE);
62 imageAttribute.setAttributeLabel("ProfileImage");
63 imageAttribute.setPicturePosition(
64     DisplayViewAttribute.PicturePosition.LEFT);
65 SingleResourceViewAttribute image = new
    SingleResourceViewAttribute(imageAttribute);
66 resourceViewAttributes.add(image);

```

```
67  
68 CardView cardView;  
69  
70 try {  
71     cardView = new CardView("Lecturer", resourceViewAttributes, name);  
72 } catch (DisplayViewException ex) {  
73     cardView = null;  
74 }  
75 [...]
```