

Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt
Fakultät Informatik und Wirtschaftsinformatik

Bachelorarbeit

Modellgetriebene Entwicklung eines Android SDKs

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

Thomas Deinlein

Eingereicht am: 11.03.2015

Erstprüfer: Prof. Dr. Peter Braun
Zweitprüfer: Prof. Dr. Oliver Hofmann

Zusammenfassung

Mobile Applikationen gibt es für eine Vielzahl von Anwendungsfällen. Der Strom an Neuentwicklungen wird aufgrund immer günstigerer mobiler Geräte und zunehmender Nutzerakzeptanz nicht abreißen. Bei der Entwicklung von Apps kommt es ganz besonders auf eine Kosten- und Zeitersparnis an: Viele Apps sind nur werbefinanziert und werden kostenlos angeboten. Nur bei sehr hohen Nutzerzahlen können Gewinne erzielt werden. Ebenso kann eine längere Entwicklungszeit dazu führen, auf dem Markt durch schnellere Konkurrenten abgehängt zu werden. Ziel dieser Arbeit ist deshalb ganz besonders zeitintensive Bereiche der Appentwicklung - Datenmodellierung, lokale Datenhaltung und Anbindung an ein Backend - durch modellgetriebene Entwicklung (MDSD) eines Android SDKs zu minimieren. Das konzipierte Android SDK soll eine vollständige Kapselung genannter Schichten ermöglichen, der/die Entwickler/in kann sich dadurch (fast) vollständig auf die Entwicklung der eigentlichen Applikation konzentrieren. Das SDK ermöglicht es Entitäten anhand einer Domain-Specific Language (DSL) zu definieren, welche automatisch zu Java-Klassen generiert werden. Objekte dieser Klassen können durch einfache Methodenaufrufe wahlfrei zentral in ein Backend, lokal in eine SQlite-datenbank oder auf kombinierte Art gespeichert und verwaltet werden.

Abstract

Mobile applications are available for a variety of use cases. The development of new applications will not be interrupted due to even cheaper mobile devices and growing user acceptance. During the process of development reducing of cost and time is important: Many apps are ad-supported or for free. Profit can be achieved only by a very high number of users. Similarly, a long development period tends to be outpaced in the app-market by faster competitors. Therefore, the aim of this work is to minimize two areas of app-development which are most time intensive - data modeling, data storage and connection to a backend - by model-driven development (MDSD) of an Android SDK. The areas are encapsulated completely by the SDK, so the developer is able to fully concentrate on programming the app. (Almost) No time has to be spent to get used to the encapsulated areas. The SDK enables the definition of entities (which are used in the app-context) by a Domain-Specific Language which are generated automatically into Java-classes. Objects of these classes can be saved (and administrated) randomly into a backend, the local SQLite-database or in a combined way by simple method calls.

Danksagung

Im Sommersemester 2014 wurde ich bereits während meiner damaligen Projektarbeit von Prof. Dr. Peter Braun betreut. Der Themenbereich umfasste bei dieser ebenso das mobile Betriebssystem Android. Die Erfahrungen, der Wissensgewinn sowie die sehr gute Betreuung empfand ich als große Bereicherung, weshalb ich mich für die Bearbeitung dieses interessanten Themengebietes entschied. Auch während meiner Bachelorarbeit stand mir Hr. Prof. Dr. Peter Braun mit Feedback und Hilfsbereitschaft jederzeit zur Seite, wofür ich mich herzlich bedanken möchte.

Auch Vitaliy Schreibmann möchte ich danken. Er unterstützte mich bei Fragen der Implementierung der DSL mit Rat und Tat zur Seite.

Ganz besonders möchte ich meiner Familie und meinen Freunden danken, die mich bereits von Beginn des Studiums an, bis auch zur Fertigstellung dieser Arbeit stets unterstützten, motivierten und mir den Rücken freihielten in zeitlich angespannten Situationen. Das durch Probelesen dieser Arbeit erhaltene Feedback war für mich eine große Hilfe, Bereicherung und Bestätigung.

Inhaltsverzeichnis

1. Einführung	1
1.1. Motivation	3
1.2. Zielsetzung	4
1.3. Aufbau der Arbeit	5
2. Grundlagen	7
2.1. Modellgetriebene Softwareentwicklung	7
2.1.1. Abgrenzung modellbasierter und modellgetriebener Softwareentwicklung	7
2.1.2. Komponenten	8
2.2. Android	12
2.2.1. Allgemeines	12
2.2.2. Nebenläufigkeit	15
2.2.3. SQLitedatenbank	18
2.3. Orca-Backend	19
2.3.1. Überblick	20
2.3.2. REpresentational State Transfer (REST)	23
2.4. Zusammenfassung	26
3. Anforderungen an das SDK	27
3.1. Allgemeine Anforderungen	27
3.2. Backend-Schnittstelle	28
3.2.1. Grundlegende Spezifikationen	28
3.2.2. HTTP-Standardmethoden	31
3.3. Lokale Datenbank-Schnittstelle	33
3.3.1. Überblick	33
3.3.2. Datenbankmethoden	33
3.4. Kombination von Backend und Datenbank	37
3.5. Zusammenfassung	40
4. Modellgetriebene Entwicklung des Android SDKs	41
4.1. Codegenerierung	42
4.1.1. DSL mit Xtext	42
4.1.2. Generator mit Xtend	44
4.2. Zusammenspiel von generiertem Code und Plattformcode	46

Inhaltsverzeichnis

4.3.	Aufbau und Struktur einer generierten Entitätsklasse	48
4.4.	Plattform-Code	49
4.5.	Orca-Backend-Schnittstelle	50
4.5.1.	Backendzugriffe mit AsyncTask	50
4.5.2.	Handhabung von Fehlern	51
4.5.3.	Verwendung von Callbacks	52
4.6.	SQLitedatenbank-Schnittstelle	53
4.6.1.	Datenbankzugriffe	53
4.6.2.	Fehlerbehandlung	58
4.7.	Sonderfall: kombinierte Schnittstelle	58
4.7.1.	Realisierung	58
4.7.2.	Workflow	60
4.7.3.	Fehlerbehandlung	60
4.8.	Qualitätssicherung	62
4.9.	Zusammenfassung	62
5.	Evaluierung der Ergebnisse	63
5.1.	Zeitersparnis	63
5.2.	Bewertung des Codeumfangs	64
5.3.	Handhabung des SDKs	67
5.4.	Modellgetriebene Entwicklung unter Android	71
5.5.	Validierung der Qualitätssicherung	71
5.6.	Fazit	72
6.	Zusammenfassung und Ausblick	74
A.	Implementierte DSL mit Xtext	77
B.	Inhalt des beigefügten Datenträgers	79
C.	Einrichtung des SDKs in eine Eclipse-Entwicklungsumgebung	80
Verzeichnisse		94
Listings		96
Literatur- und Quellenverzeichnis		99
Eidesstattliche Erklärung		100

1. Einführung

Was wären aktuelle Smartphones, Phablets oder Tablets ohne ein breites Angebotsspektrum an Apps? Es wären prinzipiell nur hochgerüstete Minicomputer, die ihr volles Potential kaum ausschöpfen könnten. Hätte sich kein so umfassender Markt an verfügbaren Apps entwickelt, wäre die Erfolgsgeschichte genannter mobiler Geräte wohl negativer verlaufen als bisher. Die Entwicklung von Apps ist somit von entscheidender Bedeutung, einerseits für Gerätehersteller, denn nur durch entsprechend aktuelle Angebote von Apps kann von einer hohen Kaufbereitschaft für jene Geräte ausgegangen werden, andererseits für Softwarehersteller, die natürlich durch immer mehr Nutzer von mobilen Geräten ebenso hohe Absätze durch Appverkäufe erzielen können. Hardware- wie auch Softwarefirmen stehen demnach in einer gegenseitigen Verpflichtung.

Grundlegend für die Entwicklung von Apps ist zunächst die Entscheidung, welches mobile Betriebssystem bzw. welche mobilen Betriebssysteme als Basis für die Entwicklung mobiler Applikationen unterstützt werden soll/en. Aus Grafik 1.1 ist die deutliche Marktführung des Betriebssystems Android von Google erkennbar, gefolgt von Apples Pendant. Beide Firmen decken demnach über 90 % des gesamten mobilen Betriebssystemmarktes ab. Es sollte deshalb selbstverständlich bei jeglicher Appentwicklung sein, beide Plattformen zu unterstützen. Appentwickler/innen stehen deshalb vor der schwierigen Aufgabe entweder zeitgleich auf beiden Plattformen zu programmieren oder nachträgliche Konvertierungen vorzunehmen. Gerade deshalb ist es wichtig, Bereiche zu kapseln, die bei jeder Appentwicklung durchlaufen werden.

Vergleicht man nun die Anzahl an verfügbaren Apps in den beliebtesten App-Stores (siehe hierzu Abbildung 1.2) könnte man zumindest bei Google Play und dem Apple App Store zu der Annahme kommen, dass bei über 1 Million Apps kein Bedarf mehr an Neuentwicklungen bestehen würde. Man könnte annehmen, dass künftig nur noch eine Anpassung der vorhandenen Apps an neue Hardware erfolgen muss, demnach hauptsächlich Wartungsarbeiten für Entwickler/innen anfallen würden. Dagegen sprechen allerdings mehrere Zukunftsprognosen.

Im Jahr 2014 besaßen weltweit rund 1,8 Milliarden Menschen ein Smartphone, im Jahr 2018 werden sogar schon 2,7 Milliarden Smartphone-Nutzer erwartet (siehe [24]). Die Entwicklung und Herstellung neuer Geräte wird demnach weiter steigen. Durch den aktuellen Trend der “Selbstoptimierung“ werden Wearables eine zunehmende Akzeptanz von Nutzern erhalten. Der Umsatz soll bis 2018 nur in Europa auf über 9 Milliarden Euro stei-

1. Einführung

gen (aktuell bei rund 4 Milliarden Euro, siehe [23]). Immer mehr „Alltagsgegenstände“ werden mit mobilen Geräten gekoppelt. Aktuelle TV-Geräte, Fahrzeuge oder gesamte Bereiche im eigenen Zuhause („Smart Home“) können per entsprechender App angesteuert und bedient werden. Gerade im letztgenannten Segment werden Umsatzverdopplungen bis 2017 auf rund 40 Milliarden US-Dollar bei der Installation und den Services erwartet (siehe [22]).

Trotz der Masse an bereits vorhandenen Apps ist somit nicht absehbar, dass künftig Neuentwicklungen ausbleiben werden.

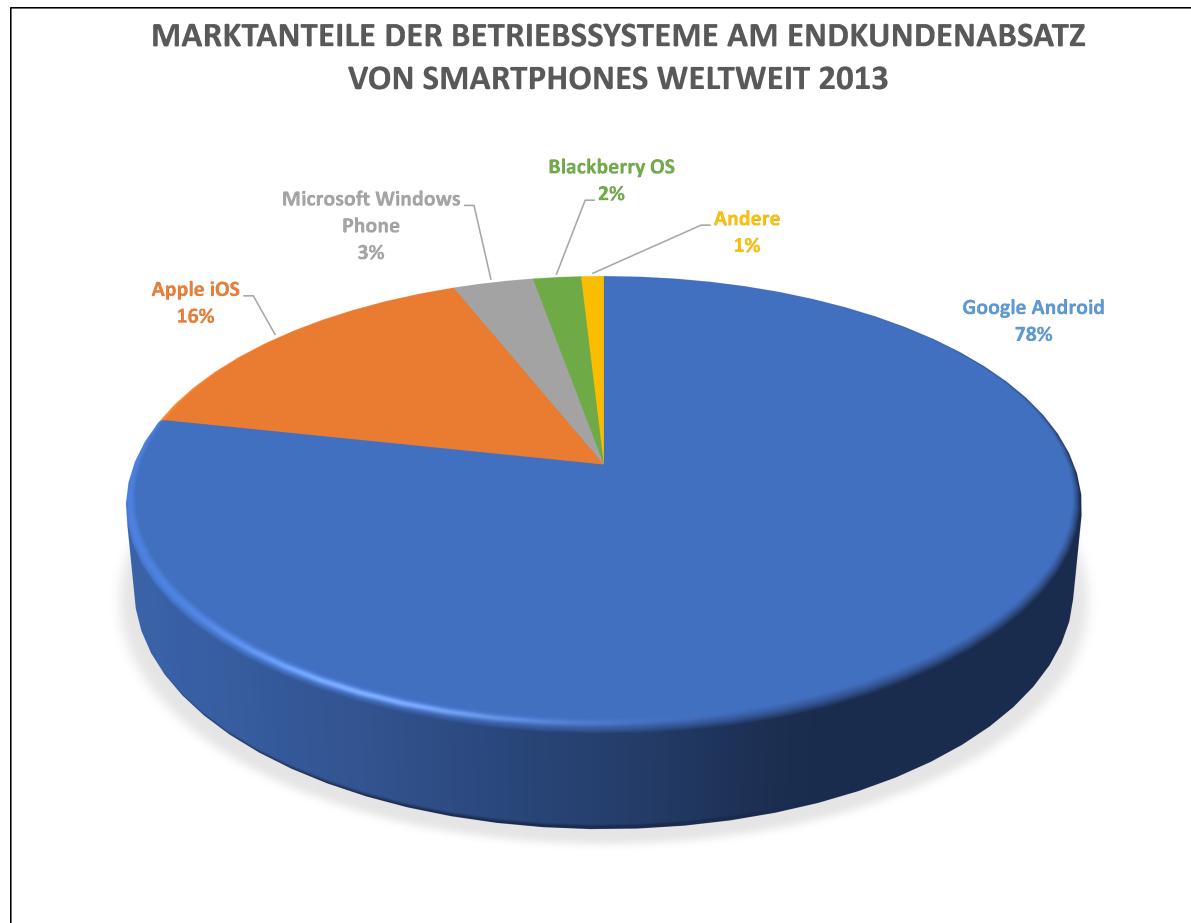


Abbildung 1.1.: Marktanteil mobiler Betriebssysteme 2013 (in Anlehnung an [21])

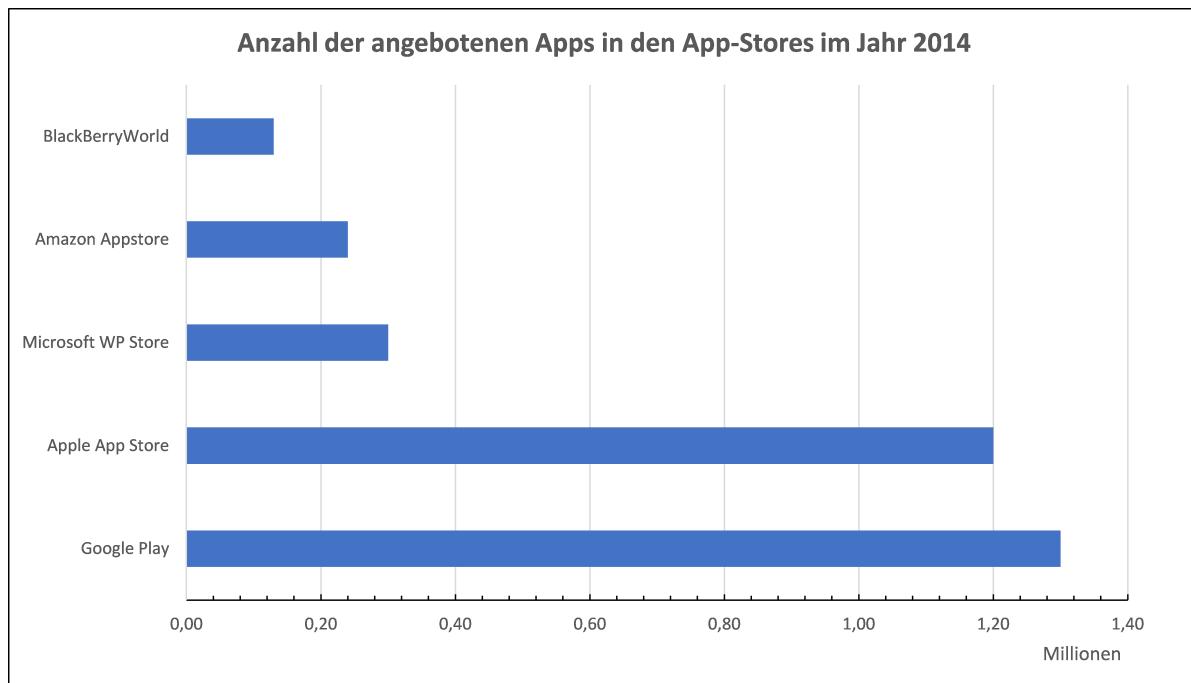


Abbildung 1.2.: Anzahl Apps in App-Stores (siehe [20])

1.1. Motivation

Wie in der Einführung belegt wurde, wird der Bedarf an der Kreation neuer und der Wartung und Verbesserung vorhandener Apps auch künftig steigen. Aufgrund der Masse an bereits vorhandenen Apps und noch hinzukommender, sowie des aufgrund des technischen Fortschritts sehr stark fragmentierten Gerätemarktes, ist eine kontinuierliche Weiterentwicklung mobiler Betriebssysteme sowie deren Apps eine herausfordernde Aufgabe.

Eine schnelle und zielgerichtete Entwicklung ist deshalb sehr wichtig, zukunftsgerichtet und kann für die Stellung des Softwareherstellers auf dem App-Markt entscheidend sein. „Doppelimplementierungen“ sollen vermieden werden und eine plattformübergreifende Entwicklung ist de-facto-Standard. Zeit und Kosten sollen bzw. müssen bei einer erfolgreichen Appentwicklung immer beachtet werden. Besonders der zeitliche Aspekt ist für den Erfolg so mancher App entscheidend.

Es stellt sich demnach die Frage, wie bei der Appentwicklung Zeit (und damit Geld) gespart werden kann. Die meisten aktuell verfügbaren Apps bieten die Möglichkeit einer persistenten Speicherung der vom Nutzer eingegebenen Daten, wobei viele Apps auf diese Funktionalitäten sogar angewiesen sind, denn z.B. eine Messenger-App wäre ohne einen Datenaustausch über ein Backend sinnlos, weil dadurch erst eine Kommunikation zwischen den Nutzern ermöglicht wird.

1. Einführung

Dafür werden deshalb Schnittstellen für die Ansteuerung geräteexterner oder gerätein-terner Komponenten wie z.B. Server oder Datenbanken benötigt. Gibt es zu der anzusteuernden Komponente keine Schnittstelle, muss neben der reinen Entwicklung der App auch noch eine - zumeist individuelle - Implementierung der Schnittstelle erfolgen. Der/die Entwickler/in muss sich deshalb entsprechend einarbeiten, was natürlich Zeit und somit (extra) Geld kostet. Erstrebenswert wäre es nun, wenn hierfür allgemeine Schnittstellen in das Entwicklungsprojekt integrierbar sind, die clientseitig angesteuert und auch bei mehreren Apps verwendet werden können. Davon würde die reine Entwick-lungszeit profitieren. Wünschenswert ist deshalb die eigentliche Appcodierung soweit wie möglich von zusätzlichen Implementierungsaufgaben zu trennen bzw. diese soweit zu ver-allgemeinern, dass dafür wiederverwendbare Schnittstellen implementiert und eingesetzt werden können.

1.2. Zielsetzung

Zum besseren Verständnis der Zielsetzung dieser Arbeit ist Abbildung 1.3 heranzuziehen. Diese Grafik stellt beispielhaft dar, welche Schichten bei der Appentwicklung auf Seiten des Frontend und Backend implementiert und miteinander verzahnt werden müssen.

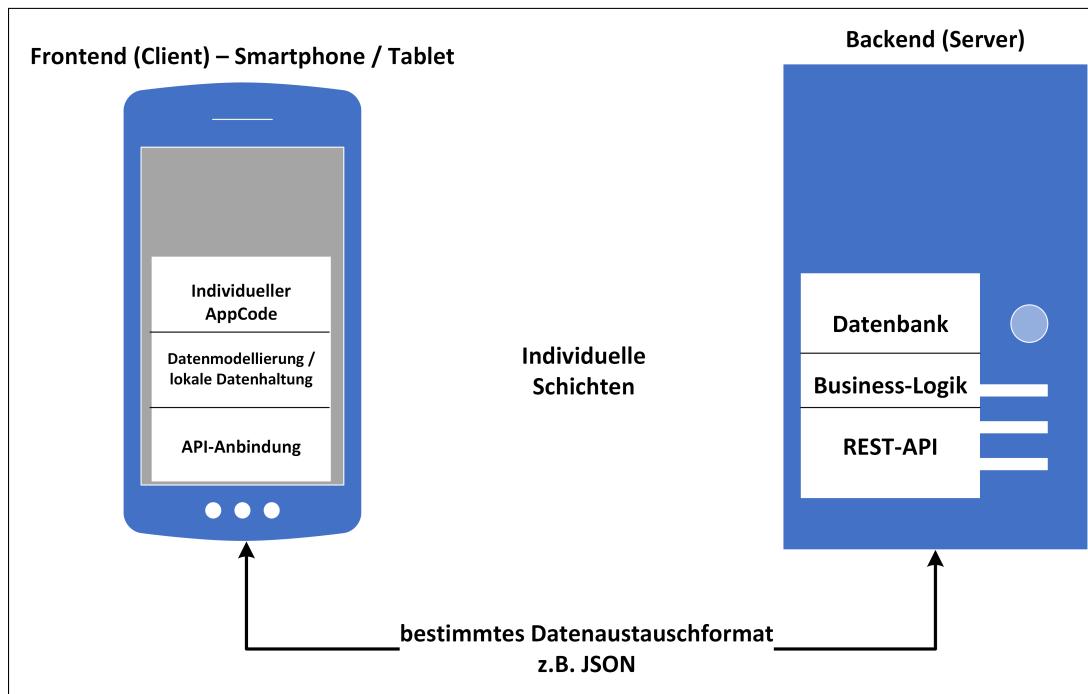


Abbildung 1.3.: Beispielhaftes Schichtenmodell zwischen Frontend und Backend (in An-lehnung an [18])

1. Einführung

Jede Schicht für sich benötigt entsprechende Einarbeitungs- und Entwicklungszeit. Besonders zeitraubend ist für den/die Appentwickler/in die Umsetzung des Datenmodells sowie die Erstellung einer Netzwerkschnittstelle für den einheitlichen Datenaustausch mit dem Backend. Hierfür sind neben entsprechender Kenntnisse auf Seiten der Appcodierung auch Implementierungskenntnisse des Backends notwendig, wodurch Zeit bei der eigentlichen Appentwicklung verloren geht. Das Datenmodell selbst ist bei vielen Apps oft einfach gehalten: Meistens werden nur wenige Entitäten benötigt, die eine bestimmte Anzahl an Attributen beinhalten. Es ist demnach meist eher klein und überschaubar. Geraade deshalb wäre es optimal, einmalig ein allgemeines abstraktes Modell zu definieren, dass für mehrere Apps verwendet werden kann.

Im Rahmen dieser Arbeit soll deshalb für das mobile Betriebssystem Android ein SDK auf Seite des Clients entwickelt werden, das es appübergreifend ermöglicht, Entitäten einerseits in einem generischen Backend zentral, andererseits mit Hilfe der standardmäßig enthaltenen SQLitedatenbank lokal persistent zu speichern, wobei dem/der Entwickler/in überlassen ist, ob er entweder das Backend, die SQLitedatenbank oder eine Kombination beider Datenhaltungskomponenten verwenden möchte. Er/Sie muss anhand einer per Domain-Specific Language definierten Modellsprache die für seine App benötigten Entitäten festlegen, woraus automatisch die entsprechenden Entitätsklassen auf Java-Ebene generiert werden. Dieser generierte Code verwendet für die Ansteuerung des Backends sowie der SQLitedatenbank das entwickelte generische SDK und ermöglicht die Nutzung von Backend und Datenbank durch einfache Methodenaufrufe auf Java-Ebene.

Der/die Entwickler/in muss sich somit kaum anderweitig einarbeiten und kann demnach erheblich mehr Zeit in die eigentliche Appcodierung investieren. Bezogen auf die in Abbildung 1.3 angesprochenen Schichten ist es Ziel dieser Arbeit, die Schicht der Datenmodellierung/lokale Datenhaltung sowie der Netzwerkschicht auf Seite des Clients vollständig durch das SDK in Kombination mit den modellgetriebenen Entwicklungswerkzeugen abzudecken. Bei der Implementierung wurde von vornherein ein besonderer Fokus darauf gelegt, möglichst viel Code generisch abzubilden, der generierte Code soll minimal gehalten werden.

Die Arbeit baut hierbei ausschließlich auf das Android-Betriebssystem auf. Vergleiche oder Beispiele zu anderen aktuellen mobilen Betriebssystemen sind nicht Bestandteil dieser Arbeit. Ebenso werden andere Softwareentwicklungsparadigmen nicht betrachtet.

1.3. Aufbau der Arbeit

Diese Arbeit gliedert sich in sechs Kapitel. Das erste Kapitel, das mit diesem Abschnitt endet, beinhaltet die Einführung in das behandelte Themengebiet, sowie die Beschrei-

1. Einführung

bung der Zielsetzung der Arbeit.

In Kapitel 2 werden Grundlagen, die für das weitere Lesen dieser Arbeit benötigt werden, erklärt. Dabei werden Grundsätze der modellgetriebenen Entwicklung, das Android-Betriebssystem sowie das Orca-Backend allgemein und zielgerichtet auf die bei dieser Arbeit involvierten Themengebiete vorgestellt. Ab diesem Kapitel wird ein fortgeschrittenes Informatikgrundwissen vorausgesetzt und nicht alle Aspekte ausführlich von Grund auf erläutert.

In Kapitel 3 wird Bezug genommen auf die Anforderungen, die das SDK genau abdecken soll. Die notwendigen Spezifikationen werden dabei für die Bereiche Backend und lokale Datenbank erörtert und festgelegt. Ebenso werden Anforderungen für eine Kombination der beiden vorherigen Funktionen festgelegt.

Kapitel 4 wird das entwickelte Framework in den wichtigsten Punkten vorstellen. Zuerst wird die Umsetzung der modellgetriebenen Softwareentwicklung ausführlich beschrieben. Danach werden der Aufbau und die Funktionsweise des SDKs bezogen auf die jeweilige Schicht vorgestellt.

Im fünften Kapitel erfolgt eine Evaluation des implementierten Frameworks. Das Kapitel analysiert hierbei den Nutzen des SDKs anhand mehrerer Aspekte. Insbesondere werden die gewünschte Zeitersparnis, der Codeumfang sowie die allgemeine Handhabung des SDKs validiert. Weitere Bestandteile dieses Kapitels bewerten die Erfahrungen, die im Rahmen dieser Arbeit mit den Komponenten und der Herangehensweise der modellgetriebenen Entwicklung gesammelt werden konnten. Abschließend erfolgt auch eine Beurteilung der Qualitätssicherung.

Den Schluss dieser Arbeit wird eine Zusammenfassung sowie künftige Aspekte des hier vorgestellten Themenbereiches ansprechen.

2. Grundlagen

In diesem Kapitel geht es um die Vorstellung wichtiger Grundlagen, die für das weitere Lesen dieser Arbeit benötigt werden. Zunächst wird erläutert, was unter modellgetriebener Softwareentwicklung verstanden wird. Es erfolgt hierbei ein Bezug auf die in dieser Arbeit verwendeten Komponenten. Des Weiteren wird kurz auf das Betriebssystem Android eingegangen. Ebenso erfolgt eine Erläuterung der wichtigsten Funktionen des von der Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt eigens entwickelten generischen Backends (Orca-Backend), auf welches die im Rahmen dieser Arbeit konzipierte Backend-Schnittstelle des SDKs abzielt.

2.1. Modellgetriebene Softwareentwicklung

Zunächst soll eine Abgrenzung Klarheit über den Begriff an sich und die Rolle des Modells im Entwicklungsprozess verschaffen. Danach werden die Komponenten einer modellgetriebenen Entwicklung vorgestellt.

2.1.1. Abgrenzung modellbasierter und modellgetriebener Softwareentwicklung

Spricht man in der Softwareentwicklung von einem Modell, geht man zumeist von einer grafischen Repräsentation und Beschreibung bestimmter Komponenten eines zu entwickelnden Softwaresystems aus. (siehe [19], Seiten 3-4) Allgemein kann in diesem Zusammenhang ein Modell auch als eine Art Dokumentation angesehen werden, sei es für die Festlegung bestimmter Spezifikationen oder für die Veranschaulichung von Funktionsweisen. Solche Modelle werden zumeist anhand der Unified Modeling Language (UML) dargestellt. Das Modell fließt dabei nicht direkt in den Programmcode mit ein, es besteht nur eine gedankliche Verbindung zur eigentlichen Softwareimplementierung. Hierbei wird von einer **modellbasierten** Form der Modellnutzung gesprochen. Diese Herangehensweise hat allerdings nachteilige Nebenerscheinungen: Bei Änderungen der Software müssen alle betroffenen Modelle angepasst werden bzw. werden Modelle häufig abgeändert, so muss diese Änderung in den Code eingearbeitet werden. Darüber hinaus

2. Grundlagen

entsteht der eigentliche Code erst durch die Interpretation der Modelle durch den/die Softwareentwickler/in.

Die **modellgetriebene** Softwareentwicklung (auch Model Driven Software Development, MDSD, genannt) involviert dagegen das Modell direkt in den Code, und die Umsetzung dieses Modells in echten Programmcode erfolgt automatisch (durch Codegenerierung). (siehe [19], Seiten 3-4) Die Namensgebung ist somit durchaus wörtlich zu verstehen: Das Modell ist im Entwicklungsprozess eine treibende Kraft. Eine Anpassung des Modells wirkt sich demnach direkt auf alle betroffenen Stellen des Quellcodes aus. Hierdurch kann enorm Zeit eingespart werden, denn es entfällt (zum Teil) die “reine“ Dokumentation und es müssen auch (fast) keine Änderungen am Modell im eigentlichen Code nachbearbeitet werden. Beschreibt das Modell einen appübergreifenden Bereich, ist auch eine Wiederverwendung des Modells in mehreren Entwicklungen möglich, was ebenso förderlich ist für die gesamte Qualität der Software sowie für deren Wartbarkeit. Durch eine Verwendung des modellgetriebenen Entwicklungsparadigmas könnte somit eine Erreichung der anfangs gesetzten Ziele der Zeit- und Kostenersparnis erfolgen.

2.1.2. Komponenten

Eine wichtige Voraussetzung für das Modell ist, dass ein bestimmter Aspekt bzw. Teilbereich der Software vollständig damit beschrieben werden muss. (siehe [19], Seiten 11-12) Selten deckt dabei das Modell 100 % der gesamten zu erstellenden Software ab. Ein Modell zielt deshalb immer auf eine bestimmte Domäne - einen fachlichen Geltungsbereich - ab. Dadurch lässt sich ableiten, dass der automatisch generierte Code (zumeist) mit manuell implementiertem (Plattform-)Code zu kombinieren ist, um die eigentliche Funktionalität der zu erstellenden Software erfüllen zu können. Für die Beschreibung und Formulierung eines solchen Modells wird eine domänenspezifische Sprache (Domain-Specific Language, künftig DSL genannt) und ein Codegenerator benötigt. Nachfolgend werden beide Komponenten anhand eines einfachen praktischen Beispiels näher erläutert.

Domain-Specific Language (DSL)

Eine DSL ist vergleichbar mit einer “normalen“ Programmiersprache (auch General Purpose Language, GPL, siehe [3], Seite 7), jedoch mit einem eindeutig abgegrenzten Anwendungsbereich - sozusagen eine “Programmiersprache für eine Domäne“ (siehe [19], Seite 30). Hiermit wird das abstrakte Modell mit all seinen möglichen Komponenten beschrieben und Syntaxregeln für die Komponentenbeschreibung festgelegt. Bezogen auf diese Arbeit ist eine solche Komponente z.B. ein `AndroidEntity`. Als DSL wird im Rahmen dieser Arbeit das Open-Source-Framework *Xtext* verwendet. Ein einfaches Beispiel einer DSL geschrieben in *Xtext* ist in Listing 2.1 zu sehen.

2. Grundlagen

Listing 2.1: Beispiel-DSL mit *Xtext* (in Anlehnung an [27])

```
1 grammar org.xtext.orcasdk.Entities with
2   org.eclipse.xtext.common.Terminals
3
4 generate entityModel
5   "http://www.xtext.org/orcasdk/entitymodel/Entities"
6
7 Model:
8   entities+=AndroidEntity*
9 ;
10
11 AndroidEntity:
12   'entity' name = ID ('extends' superType=[AndroidEntity])? '{'
13   attributes += Attribute+
14   '}'
15 ;
16
17 Attribute:
18   (type = [AndroidEntity] array?='[]'? name=ID';') | (type='string' name=ID';') | (type='boolean' name=ID';')
19 ;
```

Durch eine DSL kann nun ein bestimmtes für den jeweiligen Anwendungsfall benötigtes Domänenmodell beschrieben und festgelegt werden. Es gibt die entsprechenden Regeln für einen Parser vor. Die einzelnen Parserregeln werden dabei durch

Regelname: Regelinhalt;

festgelegt. Alle Regeln werden hierarchisch beginnend bei einem Einstiegspunkt, sozusagen der Wurzel, durchlaufen. Grafisch dargestellt würde sich eine Baumstruktur ergeben. In dem angegebenen Beispiel 2.1 wäre es nun möglich Entitäten mit all ihren enthaltenen Attributten festzulegen. Von welchem Datentyp die jeweiligen Attribute sein können, ist ebenso durch die DSL festgelegt. Alle Schlüsselwörter und -zeichen werden rot dargestellt.

Der Einstiegspunkt ist hierbei `Model`. (siehe [3], Seite 27-30) Dort wird festgelegt, dass `Model` eine Collection von Entities repräsentiert (auf die Erläuterung der Zeilen 1 und 3 wird hier nicht näher eingegangen, siehe hierzu [3] und [27]). Dies wird durch die Angabe von `+=` festgelegt. Der Stern gibt an, dass die Collection `entities` aus 0 bis n Entitäten vom Typ `AndroidEntity` bestehen kann. `AndroidEntity` erwartet zwingend das Schlüsselwort `entity`. Anschließend ist ein Name zu vergeben. Welche Zeichen ein Name enthalten darf, wird durch die Angabe `ID` geregelt (hierbei handelt es sich um eine standardisierte Parserregel, welche durch die `grammar...with`-Angabe in Zeile

2. Grundlagen

1 innerhalb dieser Grammatik verfügbar wird). Es wäre hier auch optional möglich eine Basisklasse durch das Schlüsselwort `extends` zu definieren. Durch die Klammer gefolgt von einem “?“ kann diese Angabe auch wegfallen. Der Body einer `entity` wird als Collection von `attributes` festgelegt. Dabei legt das + am Ende der Zeile fest, dass mindestens ein `Attribute` vorhanden sein muss. Jedes Attribut kann nun den Typ `string`, `boolean` oder einer zuvor festgelegten `AndroidEntity` repräsentieren. Mit dieser DSL könnte ein Domänenmodell wie in Listing 2.2 beschrieben werden.

Listing 2.2: Festlegung eines Domänenmodells anhand der Beispiel-DSL (in Anlehnung an [27])

```
1 entity person{
2     string name;
3     boolean married;
4 }
5
6 entity user extends person{
7     boolean admin;
8     string username;
9 }
```

Dieses Domänenmodell würde somit zwei Entitäten mit den Namen `person` und `user` festlegen. Ziel ist es nun, den Programmcode, der von diesen Informationen abhängig ist, automatisch generieren zu lassen. Aus o. g. Beispiel könnten somit die Java-Klassen `User` und `Person` inklusive deren Attribute und Methoden generiert werden. Um dies umsetzen zu können, wird ein Generator benötigt, der die im Domänenmodell enthaltenen Informationen automatisch auslesen und weiterverarbeiten kann.

Codegenerierung

Für die Implementierung eines Codegenerators wurde im Rahmen dieser Arbeit die Programmiersprache *Xtend* verwendet. (siehe [3], Seiten 45-69) Dieser Generator wird durch *Xtext* automatisch in der Entwicklungsumgebung Eclipse und dem *Xtext*-Projekt eingebunden. Ebenso muss der Generator nicht manuell gestartet werden. Immer wenn das Domänenmodell verändert und gespeichert wird, erfolgt automatisch die Codegenerierung. Der Generator braucht dafür im Prinzip nur zwei Angaben: Den Namen der auszugebenden Datei sowie dessen spezifischen Inhalt, der generiert werden soll (der eigentliche Java-Code als String). In Listing 2.3 ist eine mögliche Generatorklasse abgebildet. Die Methode `doGenerate(...)` bekommt zwei Objekte übergeben. `resource` beinhaltet die Informationen über das entsprechende Domänenmodell, `fsa` ermöglicht die eigentliche Codegenerierung durch Aufruf einer speziellen Methode. In der `for`-Schleife wird das `resource`-Objekt durchlaufen und es wird ein Zugriff auf alle `AndroidEntity`-Objekte möglich. `androidEntity.name.toFirstUpper` in

2. Grundlagen

Zeile 5 gibt den jeweiligen Entity-Namen zurück. Bezogen auf das Beispiel 2.2 würden die Namen `user` und `person` abgerufen. Alle weiteren Informationen können analog ausgelesen werden.

Listing 2.3: Codegenerator mit *Xtend* (in Anlehnung an [27])

```
1 Class EntitiesGenerator implements IGenerator{
2
3     override void doGenerate(Resource resource, IFileSystemAccess
4                               fsa) {
5
6         for(androidEntity:
7             resource.allContents.toIterable.filter(AndroidEntity)) {
8
9             fsa.generateFile(androidEntity.name.toFirstUpper+".java",
10                  CreateAndroidEntity.compile(androidEntity));
11         }
12     }
13 }
```

Der eigentliche Generator ist der Aufruf von `fsa.generateFile(..., ...)`. (siehe [3], Seite 88-89) Durch diesen Aufruf wird für alle in der `resource` enthaltenen `AndroidEntity` eine Datei mit Namen der Entity (erster Buchstabe wird groß geschrieben) gefolgt von `".java"` und der Zeichenkette, die die statische Methode `compile` der Klasse `CreateAndroidEntity` zurückgibt, erstellt. Innerhalb dieser Methode können auch die entsprechenden Informationen anhand des übergebenen `androidEntity`-Objektes ausgelesen und individuell zu einer Zeichenkette zusammengefügt werden. Die `compile`-Methode könnte wie in Listing 2.4 implementiert werden.

Listing 2.4: Beispiel-Template in *Xtend* (Quelle: Eigene Darstellung)

```
1 def static compile(AndroidEntity entity) {
2
3     ''
4     public <<entity.type.toFirstUpper>> get<<entity.name.toFirstUpper>>()
5     {
6         return <<entity.name>>;
7     }
8
9     ''
10 }
```

Der Code, der zwischen den drei einfachen Anführungszeichen geschrieben ist, wird als Zeichenkette zurückgegeben. (siehe [26]) In diesem Fall wird eine Getter-Methode generiert. Auffällig sind hierbei auch die französischen Anführungs- und Schlusszeichen

2. Grundlagen

<< und >>. Hierüber können dynamisch Methoden und Attribute des eigentlichen *Xtend*-Codes verwendet werden. Dies ist für die Codegenerierung von entscheidender Bedeutung. Darüber können die im Domänenmodell eingegebenen Informationen in das Codegenerat übernommen werden. <<entity.name>> liest somit den Namen der Entität aus dem Domänenmodell und “druckt“ diesen in den zu generierenden Code an die Stelle, an der dieser Aufruf erfolgt.

Das Zusammenspiel von DSL, Domänenmodell und Generator ist zum einfacheren Verständnis in Abbildung 2.1 zusammenfassend grafisch dargestellt.

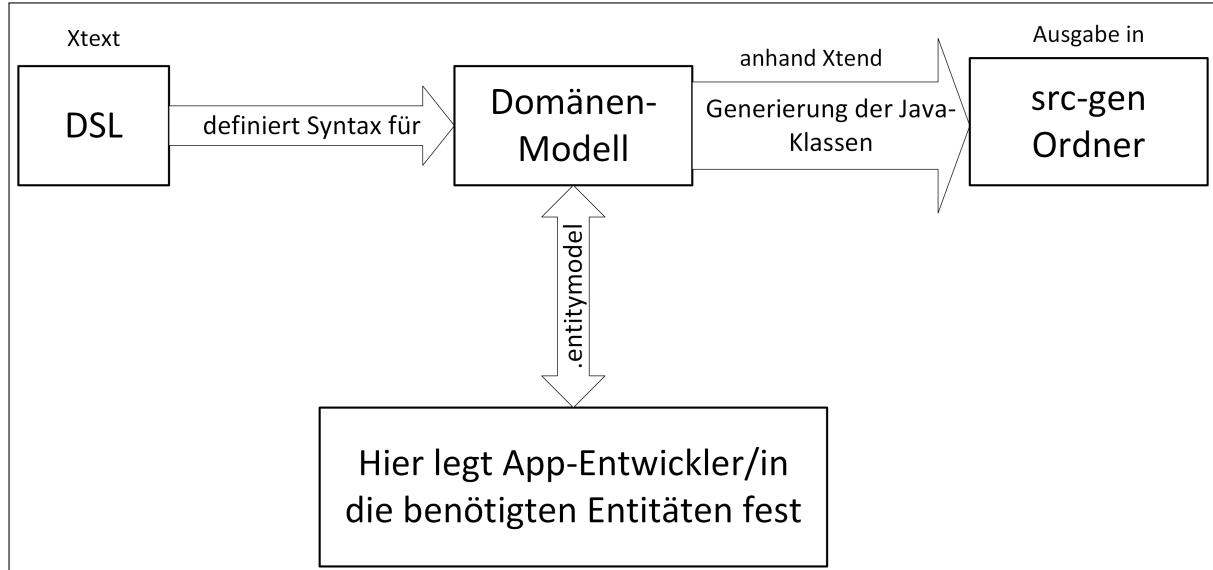


Abbildung 2.1.: Zusammenspiel von DSL, Domänenmodell und Generator (Quelle: Eigene Darstellung)

2.2. Android

Dieses Kapitel beschreibt nur die für diese Arbeit relevanten Bereiche des Android-Betriebssystems. Neben einem allgemeinen Überblick wird besonders auf Nebenläufigkeit und SQLitedatenbanken eingegangen.

2.2.1. Allgemeines

Android ist ein mobiles Betriebssystem, das ursprünglich für Mobiltelefone konzipiert war. Mittlerweile wird es allerdings auf verschiedenen Gerätetypen eingesetzt. Demnach muss es eine Bandbreite an Funktionen bereitstellen, die diese Typenvielfalt be-

2. Grundlagen

herrschbar macht. Anwendungen werden in der Programmiersprache Java geschrieben. In Abbildung 2.2 ist die Systemarchitektur Androids grafisch dargestellt.

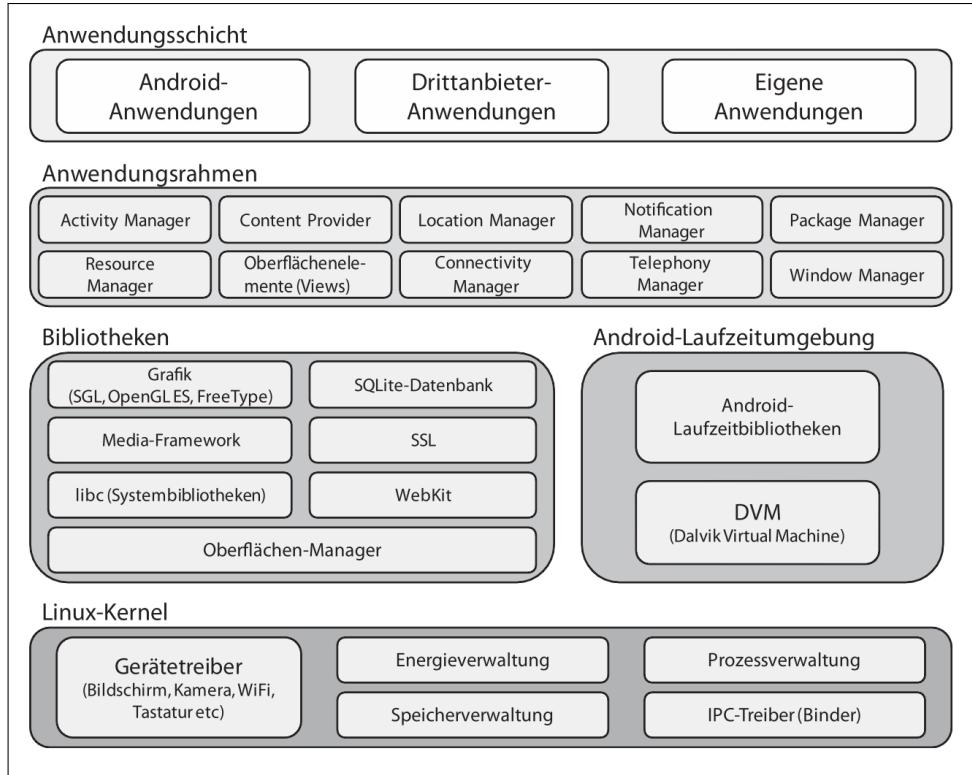


Abbildung 2.2.: Systemarchitektur von Android (siehe [2], Seite 25)

Dalvik Virtual Machine

Insbesondere die Laufzeitumgebung ist dafür verantwortlich, dass sich Portierungen auf verschiedenste Hardwarearchitekturen komfortabel durchführen lassen. (siehe [2], Seite 26-28) Den Kern der Laufzeitumgebung bildet die Dalvik Virtual Machine, künftig DVM genannt. Pro Anwendung wird ein eigener Betriebssystemprozess gestartet und innerhalb dieses Prozesses eine eigene DVM. Diese unterscheidet sich von einer klassischen Java Virtual Machine (JVM). Eine JVM nutzt Mikroprozessorregister nicht aus, eine DVM dagegen schon. Berechnungen von Zwischenergebnissen werden dadurch stark beschleunigt. Eine DVM arbeitet nach dem Prinzip einer Registermaschine, eine JVM wie eine Kellermaschine. Damit eine DVM eine Android-Anwendung ausführen kann, muss diese in ein spezielles Bytecode-Format umgewandelt werden. Hierfür muss zuerst eine Umwandlung in normalen Java-Bytecode wie bei einer JVM erfolgen. Danach erfolgt durch ein spezielles Tool (dx-Tool) eine Übersetzung in den Dex-Bytecode. Dieser kann von einer DVM verarbeitet werden. Für eine übersichtlichere Darstellung ist Abbildung 2.3 heranzuziehen.

2. Grundlagen

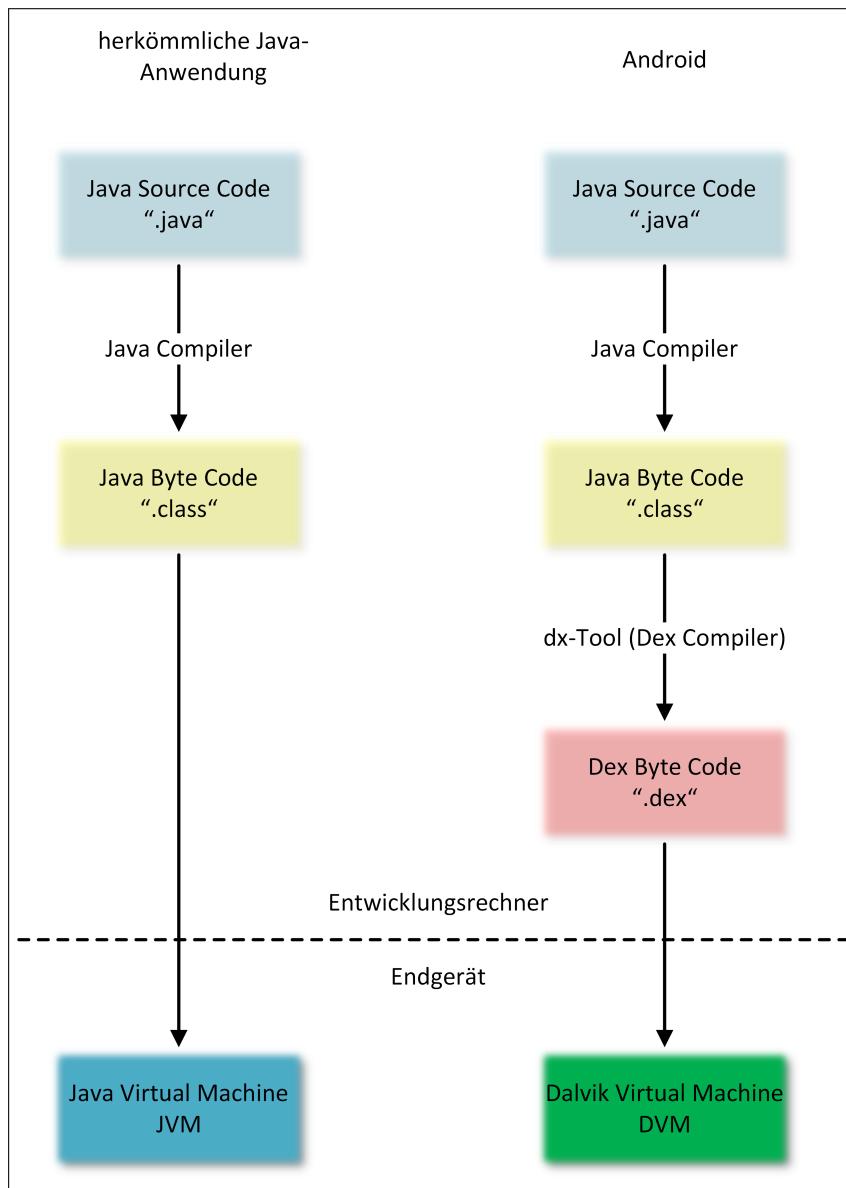


Abbildung 2.3.: Vergleich von JVM und DVM (in Anlehnung an [2], Seite 28)

Hardwareherstellern wird es auch ermöglicht, die (quelloffene) DVM so anzupassen, dass diese in der Lage ist entsprechenden Maschinencode zu produzieren, der von eigenen speziellen registerbasierten Prozessoren verarbeitet werden kann. Dadurch wird der Aufwand für Portierungen verringert.

Anmerkung: Durch Einführung von Android 5.0 wurde die Dalvik Virtual Machine durch die *Android Runtime (ART)* ersetzt. Die sich dadurch ergebenden Änderungen werden im Rahmen dieser Arbeit nicht erläutert, da während der gesamten Erstellungsphase des SDKs ein Smartphone mit Android 4.3 verwendet wurde. Informationen zur ART sind

2. Grundlagen

unter [7] abrufbar.

Activity

Apps fallen in den Bereich der Anwendungsschicht. Diese greift auf Schnittstellen innerhalb des Anwendungsrahmens zu. In diesem Rahmen ist eine Komponente für die App-Programmierung unter Android von sehr großer Bedeutung: Die Komponente `Activity`. Eine `Activity` ist für die grafische Gestaltung einer Bildschirmseite einer App sowie deren Funktionalität zuständig, wie z.B. Behandlung von Anwendereingaben oder welche Oberflächenelemente angezeigt werden sollen. (siehe [2], Seite 31) Jede Bildschirmseite besteht üblicherweise aus einer separaten `Activity`. Eine Anwendung (App) besteht zumeist aus mehreren entsprechend verknüpften Activities (muss dies allerdings nicht zwangsweise, es gibt auch Anwendungen, die komplett ohne Activities auskommen, auf jene wird hier jedoch nicht weiter Bezug genommen).

Wie bereits erwähnt, wird jede Anwendung in einer eigenen DVM gestartet. Die Anwendung selbst läuft innerhalb der DVM in einem einzelnen Thread, dem sogenannten Main Thread. Dieser ist für alle anfallenden Aufgaben einer Activity (Oberflächendarstellung und Nutzereingaben) verantwortlich. (siehe [2], Seite 168-170) Die Ressourcen des Main Threads sollten deshalb sparsam eingesetzt werden, damit dieser nicht blockiert. Eine Blockierung würde entstehen, wenn auf dem Main Thread ein langlaufender aufwändiger Programmteil ausgeführt werden würde. Erst wenn dieser abgearbeitet wurde, „kümmert“ sich der Main Thread um die weiteren auszuführenden Aufgaben. Eine Blockierung würde schlimmstenfalls zu einem „Einfrieren“ der Benutzeroberfläche führen, der Nutzer hätte keine oder nur eine eingeschränkte Möglichkeit mit der App zu interagieren (siehe [13], Seite 121).

Für heutige Apps wäre ein (ständiges) Einfrieren des Bildschirms ein K.o.-Kriterium. Langlaufende Aufgaben sind deshalb in eigene Threads auszulagern. Ein Beispiel für einen langlaufenden Programmteil sind Netzwerzkzugriffe auf ein Backend.

2.2.2. Nebenläufigkeit

Die manuelle Verwendung und Steuerung von separaten Threads im Zusammenhang mit Netzwerzkzugriffen ist - wie im vorherigen Abschnitt schon beschrieben - von sehr großer Bedeutung. Die im Rahmen dieser Arbeit konzipierte Backend-Schnittstelle muss demnach gewährleisten, dass bei einer Verwendung durch den/die App-Entwickler/in alle Netzwerzkzugriffe auf einem separaten Thread ausgeführt werden.

AsyncTask

Von Android wird die abstrakte Klasse `AsyncTask` zur Verfügung gestellt, die für die Abarbeitung langlaufender Programmteile in einem eigenen Thread ausgelegt ist. (siehe [13], Seite 126-130) Die Backend-Schnittstelle verwendet bei allen Netzwerkzugriffen diese Klasse, weshalb die wichtigsten Bestandteile hier aufgeführt werden.

Um die `AsyncTask`-Funktionalitäten nutzen zu können, muss eine eigens erstellte Klasse geschrieben werden, die von `AsyncTask` ableitet. (siehe [13], Seite 126-130) `AsyncTask` ist eine generische Klasse und benötigt drei generische Parameter. Der erste Parameter gibt den Datentyp der Parameter an, die an den separaten Thread übergeben werden. Dieser erste Parameter ist beim Start des `AsyncTask` zu übergeben. Ein `AsyncTask` wird durch Aufruf der `execute`-Methode gestartet, die nach der Instantiierung auf diesem `AsyncTask`-Objekt aufzurufen ist. Dieser Methode ist ein Parameter oder ein Array von Objekten von dem Datentyp des ersten generischen Parameters zu übergeben.

Der mittlere Parameter ist für Statusupdates auf dem aufrufenden Thread zuständig. (siehe [13], Seite 126-130) In allen verwendeten `AsyncTask`-Klassen im Rahmen dieser Arbeit wurde dieser Parameter nicht verwendet und immer auf `Void` gesetzt. Von einer weiteren Erläuterung wird deshalb abgesehen. Weiterführende Informationen hierzu sind unter [6] verfügbar.

Der letzte Parameter ist der Datentyp des Wertes, der an den aufrufenden Thread nach erfolgter Abarbeitung zurückgegeben wird. (siehe [13], Seite 126-130) Für alle drei Parameter gibt es spezielle Methoden, die innerhalb der `AsyncTask`-Klasse zu überschreiben sind. Die wichtigste Methode ist `doInBackground (params...)`. Das ist die Methode, die auf dem separaten Thread ausgeführt wird. Hier sind die entsprechend langwierigen Programmteile zu implementieren. Ist die Abarbeitung beendet, wird das Ergebnis an die Methode `onPostExecute (... result)` übergeben. Diese Methode wird auf dem aufrufenden Thread (normalerweise der Main Thread) ausgeführt. Als Übergabeparameter erhält diese den Rückgabewert der `doInBackground`-Methode. Beispiel: Eine Klasse mit `AsyncTask<Integer, Void, String>` als Basisklasse würde `Integer`-Parameter bei der `execute`-Methode erwarten. Somit würde die `doInBackground`-Methode `Integer`-Objekte als Parameter erhalten, diese verarbeiten und ein Objekt vom Typ `String` zurückgeben. Die Methode `onPostExecute` bekommt diesen `String` übergeben. Der langlaufende Prozess ist hier schon abgearbeitet und eine Verwendung des `String` ist ohne unangenehme Seiteneffekte möglich.

Die Verwendung von `AsyncTask` ist sehr vorteilhaft, denn es ist deutlich abgegrenzt und erkennbar, welcher Code auf einem separaten Thread ausgeführt wird (`doInBackground`-Methode). Ebenso erfolgt auch ein “Hinweis”, wenn der separate Thread seine Aufgabe abgearbeitet hat (Aufruf der `onPostExecute`-Methode). Zum leichteren Verständnis

2. Grundlagen

des Datenflusses ist Abbildung 2.4 heranzuziehen.

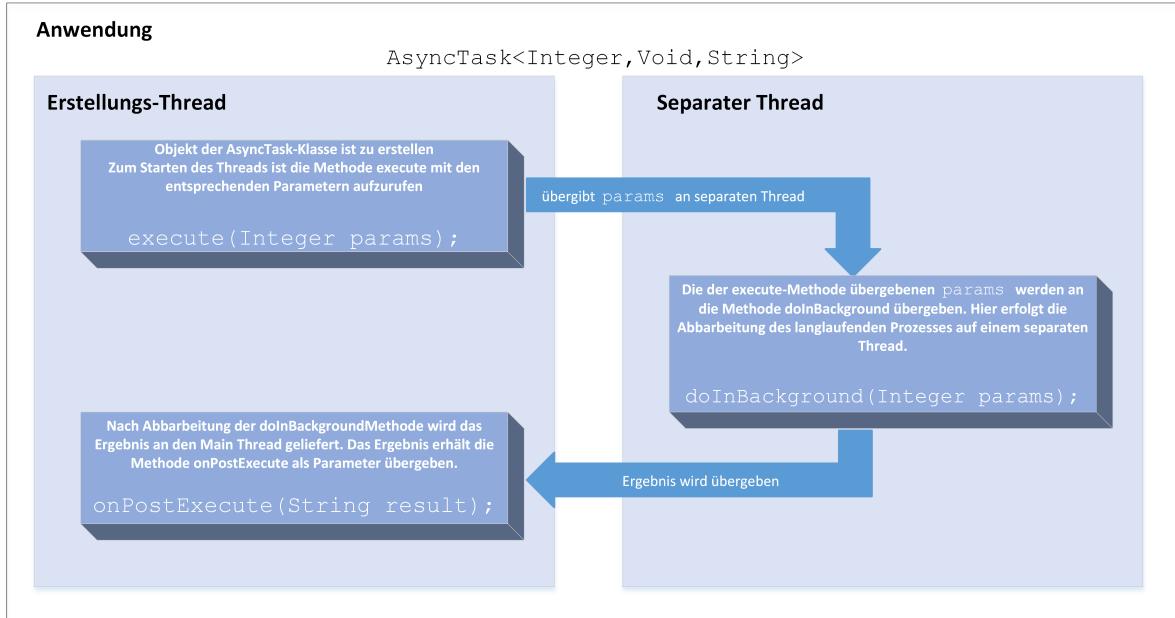


Abbildung 2.4.: Datenfluss einer AsyncTask-Klasse (in Anlehnung an [13], Seite 127)

Anmerkung: Es können auch Parameter über einen entsprechenden Konstruktor übergeben werden. Der eigentliche Übergabemechanismus kann somit auch ungenutzt bleiben.

Callback

Die Verwendung von AsyncTask-Klassen hat allerdings auch Nachteile. Verwendet man den Standard-Rückgabeparameter-Mechanismus, ist man deutlich eingeschränkt, schließlich können damit nur Parameter eines(!) Datentypes an den Main Thread zurückgegeben werden. Gerade bei Netzwerkzugriffen ist es allerdings durchaus notwendig StatusCodes (Typ int) und Fehlermeldungen (Typ String) nach dem (versuchten) Netzwerkzugriff abfragen zu können. Die innerhalb der Backend-Schnittstelle verwendeten AsyncTask-Klassen wurden deshalb für eine Kombination mit einem *Callback* entworfen.

Bei allen *Callbacks* handelt es sich um Interface-Klassen mit einer einzigen `onComplete`-Methode. Je nach benötigten Rückgabewerten werden individuelle *Callbacks* verwendet. Durch das *Callback* kann sichergestellt werden, dass die entsprechende Anfrage ans Backend erfolgt und abgeschlossen ist. Ebenso können dadurch Rückgabeparameter threadsicher abgefragt werden. Dies wird durch Aufruf der `onComplete`-Methode innerhalb der `onPostExecute`-Methode der AsyncTask-Klasse sichergestellt. Das *Callback*

2. Grundlagen

ist bei Aufruf einer entsprechenden Backend-Methode als Objekt einer anonymen Klasse zu übergeben. In Unterabschnitt 4.5.3 wird hierauf noch genauer eingegangen und ein Beispiel aufgeführt.

2.2.3. SQLitedatenbank

Eine Komponente des Android-Betriebssystem ist das Datenbanksystem SQLite. Jede Anwendung kann dieses für seine eigenen Zwecke verwenden. Es bietet sich daher für eine Anwendung an, die vom Backend angeforderten Daten lokal zwischenspeichern und erst nach erfolgter lokaler Bearbeitung im Backend zu aktualisieren. Ebenso würde sich eine autarke Nutzung als lokaler Speicher für wichtige Daten einer App anbieten. Da das SDK dieses Datenbanksystem involviert, werden grundlegende Funktionen in dem folgenden Abschnitt erläutert.

Relationale Datenbank

Bei dem lokalen Datenbanksystem handelt es sich um eine relationale Datenbank. (siehe [11], Seite 301-302) Daten werden demnach in Tabellen abgelegt. Dabei spiegeln die Spalten der Tabelle die entsprechenden Attribute und die Zeilen die einzelnen Datensätze wieder. Daten verschiedener Tabellen können sich gegenseitig referenzieren und somit Beziehungen aufweisen. Es bedarf keiner gesonderten Installation. (siehe [2], Seite 246) Die Datenbank steht einer Anwendung ohne explizite Startbefehle zur Verfügung. Innerhalb des Android-Dateisystems wird für jede Datenbank eine Datei angelegt. Dabei ist der Pfad einheitlich. Alle Datenbanken werden im Verzeichnis `data/data/{packageNameDerAnwendung}/databases/{Datenbankname}` (die geschweiften Klammern sind kein Bestandteil des eigentlichen Pfades) angelegt. Der Datenbankname muss innerhalb einer Anwendung dabei immer eindeutig sein.

Anlegen, Verwalten und Handhabung der Datenbank

Für das Anlegen einer Datenbank wird eine Klasse benötigt, die von der Klasse `SQLiteOpenHelper` abgeleitet ist. (siehe [11], Seite 303) Innerhalb des Konstruktors ist der Datenbankname festzulegen. Dabei wird automatisch geprüft, ob es diese Datenbank bereits gibt oder diese erst erstellt werden muss. Bei einer Neuanlegung wird immer die `onCreate`-Methode aufgerufen. In dieser Methode müssen nun alle benötigten Tabellen erstellt werden. Das bedeutet, dass das Tabellschema der zu speichernden Daten hier bereits festgelegt sein muss. Um das Schema definieren zu können, müssen die Spaltennamen (Attribute) sowie deren Datentypen bekannt sein.

2. Grundlagen

Dem Konstruktor kann noch eine weitere Information mitgegeben werden: Eine ganzzählige Versionsnummer. (siehe [11], Seite 305) Anhand dieser könnte ein Aufruf der `onUpgrade`-Methode erfolgen. Dies würde geschehen, wenn bereits eine entsprechende Datenbank mit einer kleineren Versionsnummer bei Aufruf des Konstruktors vorhanden ist. Alle im Rahmen dieser Arbeit konzipierten Datenbank-Schnittstellen machen von diesem Mechanismus allerdings keinen Gebrauch. Die SQLitedatenbank soll im Rahmen des SDKs für eine Art Zwischenspeicher verwendet werden können und die grundlegenden Datenbankfunktionen abdecken.

Nach allen Datenbank-Transaktionen ist die Datenbank per `close()`-Methode zu schließen (siehe [2], Seite 251).

Bei Datenbankanfragen wird vom System ein Objekt vom Typ `Cursor` zurückgegeben. Darin enthalten sind alle gefundenen Datensätze in Form einzelner `Rows` (die jeweiligen gefundenen Zeilen der durchsuchten Tabelle). `Cursor`-Objekte können aus mehreren `Rows` bestehen. Der `Cursor` ist `Row` für `Row` zu durchlaufen. Jede `Row` kann dabei als eine Art `HashMap` angesehen werden. Um allerdings auf die einzelnen Attribute (Spalten) der jeweiligen `Row` zugreifen zu können, ist die Kenntnis der Spaltennamen sowie deren Datentypen erforderlich.

Es ist für die Erstellung und Handhabung der SQLite-Datenbank somit immer erforderlich, die entsprechende Datenstruktur (deren Attribute und deren Datentypen) zu kennen. Dadurch wird eine generische Implementierung einer Klasse für die Verwaltung der Datenbankzugriffe erschwert.

2.3. Orca-Backend

Die konzipierte Backend-Schnittstelle nutzt das von der Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt eigens entwickelte Orca-Backend. Hierbei handelt es sich um ein generisches Backend, das nach den REST-Prinzipien konstruiert wurde (siehe [16]). In diesem Abschnitt werden grundlegende Funktionen für das bessere Verständnis der Schnittstellen-Implementierung erläutert. Weiterführende Informationen können in der Dokumentation unter [16] nachgeschlagen werden.

2.3.1. Überblick

Einführungsbeispiel

Zum besseren Verständnis und Überblick soll durch eine Beispielgrafik die grundlegende Funktionsweise des Orca-Backend OHNE das entwickelte SDK grafisch veranschaulicht werden. In Abbildung 2.5 sind auf einem mobilen Gerät zwei Apps installiert, die App *Fahrtenbuch* und die App *Messenger*. Beide Apps nutzen das Orca-Backend. Bei der *Fahrtenbuch*-App wurde nun z.B. eine Entität *Fahrzeug* festgelegt. Um diese im Orca-Backend zu speichern, muss nun zunächst eine Umwandlung des Java-Objektes in das JSON-Format erfolgen (siehe Datentypen). Der Appname und Apikey muss in den Authorization-Header des HTTP-Requests übernommen werden (siehe Autorisierung) und für alle Entitäten müssen eindeutige EntityTypes (Long-Werte) gesetzt werden. Bei einer erfolgreichen Speicherung vergibt das Orca-Backend eine eindeutige ID und URI für die jeweilige Entität. Die Vorgehensweise bei der *Messenger*-App zur Speicherung einer User-Entität ist analog zu sehen.

Generizität

Generisch bedeutet im Rahmen des Backends, dass es Entitäten aufnehmen kann, die unterschiedliche Attribute besitzen. (siehe [14]) Entscheidend hierbei ist, dass die jeweiligen Entity-Attribute als Key-Value-Paar abgebildet werden. Ein Key muss dabei immer vom Typ String sein, ein Value kann dagegen verschiedene Datentypen haben (siehe Datentypen). Das Orca-Backend ist demnach eine *schemafreie dokumentenorientierte Datenbank*. Attribute werden künftig auch als Properties bezeichnet, womit immer eine Sammlung von Key-Value-Paaren gemeint ist (in Java repräsentiert das ein `HashMap<String, Object>`-Objekt). Es ist somit nicht notwendig ein Schema für jeden Entity-Typ festzulegen, um Entitäten abzuspeichern. Unterschiedliche Entitäten werden nur anhand eines von dem/der App-Entwickler/in festgelegten Zahlenwertes vereinheitlicht (`type`, siehe Datentypen). Die eigentliche Datenstruktur der Entitäten muss somit - nicht wie bei relationalen Datenbanken, siehe Unterabschnitt 2.2.3 - vorher festgelegt werden.

Datenaustauschformat zwischen Backend und Applikationen

Damit das Backend die übergebenen Entitäten verarbeiten kann, sind diese in einem einheitlichen Datenformat zu übergeben. (siehe [14]) Hierfür wird das JSON-Format (Java Script Object Notation) verwendet. Es ermöglicht auf verhältnismäßig einfache Weise Objektstrukturen als einfache String-Zeichenketten darzustellen. Ebenso gibt es be-

2. Grundlagen

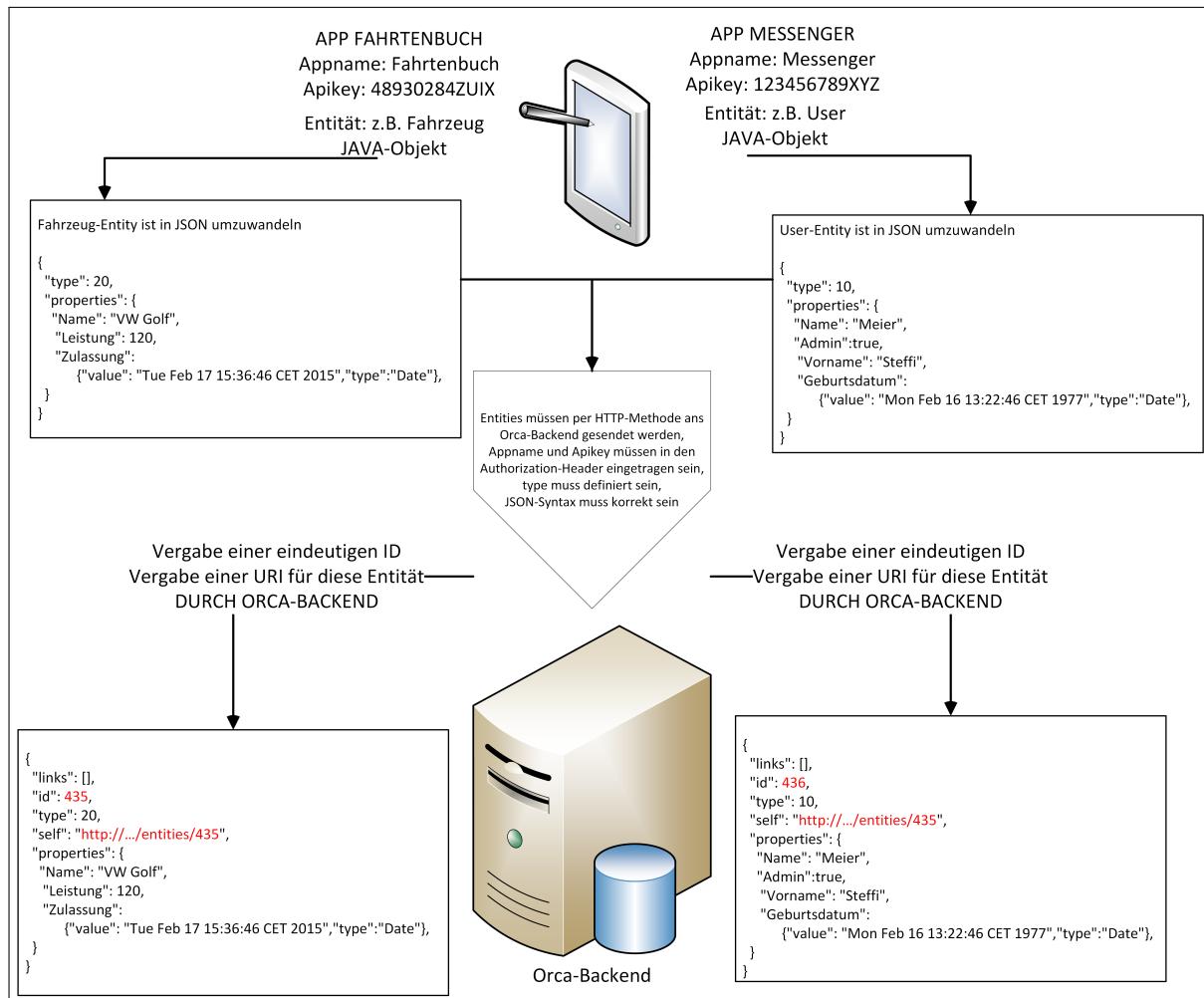


Abbildung 2.5.: Funktionsweise des Orca-Backend OHNE das SDK (Quelle: Eigene Darstellung)

2. Grundlagen

reits mehrere Frameworks, die die Übersetzung eines Java-Objektes in ein JSON-Objekt durchführen. Auf Seiten des Backends sowie der konzipierten Schnittstelle wird das *Genson*-Framework verwendet (nähtere Informationen dazu sind unter [5] abrufbar).

Datentypen und JSON-Syntax

Für die Properties dürfen allerdings nur bestimmte Datentypen verwendet werden. Es können nur folgende einfache Datentypen vom Orca-Backend verarbeitet werden: Boolean, Long, Double, String (siehe [14]). Zusätzlich ist es möglich, Koordinaten in Form eines Double-Arrays sowie Datums-Werte mit einem bestimmten Date-Format anzugeben. Locations müssen demnach immer Double-Arrays mit genau zwei Double-Werten (Längen- und Breitengrad) sein. Ein Datum muss immer im Format “EEE MMM dd HH:mm:ss zzz yyyy“ angegeben werden. Eine Entity im JSON-Format muss eine einheitliche Form besitzen, um die vom Backend vorgegebene Syntax einzuhalten. In Listing 2.5 ist eine Beispiel-Entity abgebildet.

Listing 2.5: Beispiel für ein JSON-Entity (in Anlehnung an [14])

```
1 {
2     "links": [],
3     "properties": {
4         "timestamp": {"value": "Tue Feb 17 15:36:46 CET 2015
5         , "type": "Date"}, 
6         "location": [44.3, 9.88],
7         "name": "Deinlein",
8         "firstname": "Thomas"
9     },
10    "self": "STRING_MIT_SELF_URL",
11    "type": 20
12    "id": 701
13 }
```

In dem Beispiel-Code sind neben den Properties noch weitere Attribute eines zum Orca-Backend kompatiblen JSON-Entity-Objektes erkennbar. Von entscheidender Bedeutung sind `id` und `type`. Die `id` ist die eindeutige Entity-Id, die beim Speichern vom Orca-Backend vergeben wird. Der `type` ist manuell festzulegen und kennzeichnet Entitäten des gleichen Typs. Dies ist mit einem eindeutigen Entity-Namen (Klassen-Name) vergleichbar. `type` muss allerdings vom Datentyp `Long` sein. `self` ist eine URL, über die diese Entität erreichbar ist. Diese wird bei erstmaliger Speicherung vom Orca-Backend vergeben.

Weitere Datenstrukturen: Link, EntityPage, LinkPage

Das Orca-Backend verwendet für die Durchführung seiner Funktionen noch weitere Datenstrukturen. Jede Entität kann neben dem *Link* auf seine eigene Ressource (Self-URI) auch noch *Links* zu anderen Ressourcen (anderen Entitäten) haben. Ein *Link* ist dabei kein eigenständiges Objekt, sondern immer an die entsprechende Entität gebunden. Die Entität ist dabei der Knoten, der *Link* die gerichtete Kante, die auf eine andere Entität zeigt. Ein *Link* hat in seiner JSON-Repräsentation ebenso mehrere Attribute: `object` ist hierbei die eigene URI, `type` ist der LinkType als String. Ebenso hat ein *Link* Properties. Diese sind ebenso wie bei den Entitäten frei wählbare Key-Value-Paare mit den selben erlaubten Datentypen.

Das Orca-Backend ermöglicht auch die Abfrage von mehreren Entitäten und *Links*. Diese Collections werden in JSON als *EntityPage* bzw. *LinkPage* repräsentiert. Beide PageTypen integrieren ein Array mit Entitäten bzw. Links. Hinzu kommen noch NavigationLinks, die PageSize für die Anzahl an abgefragten Entitäten bzw. Links und die gesamte Anzahl an Entitäten bzw. Links, die zu dieser Anfrage im Backend vorhanden sind (siehe Paginierung in Abschnitt 2.3.2 und [14]).

Autorisierung

Für die Autorisierung und Verwendung des Orca-Backend ist es entscheidend einen eindeutigen *Appnamen* sowie den dafür vergebenen *Apikey* im Authorization-Header der jeweiligen Anfrage einzutragen. Bei entsprechender Verwendung der im Rahmen dieser Arbeit konzipierten Lösung sind beide Angaben nur einmal an zentraler Stelle durchzuführen.

2.3.2. REpresentational State Transfer (REST)

REST ist ein Architekturstil, der erstmalig im Jahre 2000 von R.T. Fielding in dessen Dissertation (siehe [17], Seiten 77-86, die unten genannten Constraints beziehen sich hierauf) beschrieben wurde. Nach diesem Architekturstil wurde das Orca-Backend konzipiert. Zunächst werden die ursprünglich von Fielding formulierten Constraints erläutert. Da aktuelle REST-Architekturen mittlerweile auch weitere bzw. andere Aspekte erfüllen müssen und das Orca-Backend auch nach aktuellen Anforderungen konzipiert wurde, wird auch auf diese Bezug genommen.

Constraints nach Fielding

Fielding geht bei der Beschreibung der Voraussetzungen von REST zunächst von einem *Null-Style* aus, welchem Stück für Stück Constraints (Voraussetzungen) hinzugefügt werden.

Der **Client-Server-Constraint** ist die erste Voraussetzung. Fielding begründet diesen Constraint durch mehrere Argumente. Hierdurch erfolgt eine klare Trennung der Aufgabenbereiche zwischen Datenrepräsentation (user interface) und Datenverwaltung (data storage). Dadurch wird die Portierung der Datenrepräsentation auf mehrere Plattformen vereinfacht. Die Skalierbarkeit verbessert sich durch Vereinfachung der serverseitigen Komponenten. Aufgrund der Trennung können die Komponenten unabhängig voneinander entwickelt werden.

Zustandslosigkeit (*Stateless*) wird als nächste Voraussetzung von Fielding genannt. Damit ist gemeint, dass jede Anfrage des Clients an den Server alle Informationen enthalten muss, die für die Verarbeitung der Anfrage benötigt werden. Die Zustandsinformationen verwaltet dabei ausschließlich der Client. Vorteile hiervon sind eine verbesserte Skalierung (Anfragen sind nicht an Server gebunden, keine Speicherung von Sessions) und Zuverlässigkeit (Seiteneffekte bleiben aus, Fehleranfälligkeit sinkt). Ebenso wird Transparenz erreicht: Der Server muss nur eine einzige Anfrage beachten. Nachteilig ist allerdings die erhöhte Netzlast, da pro Anfrage mehr Overhead produziert werden muss.

Als **Cache**-Constraint wird eine weitere Voraussetzung bezeichnet. Alle Daten einer Antwort auf eine Anfrage müssen als *cacheable* oder *non-cacheable* gekennzeichnet werden. Der Client ist hierbei für die Verwaltung eines lokalen Caches verantwortlich. Als *cacheable* gekennzeichnete Daten dürfen bei erneutem Bedarf wiederverwendet werden. Dieser Constraint ist für die Reduzierung der Netzlast angedacht. Vorteile sind eine Reduzierung der zu übertragenden Datenmenge, Erhöhung der Performance und eine Verbesserung der Skalierbarkeit. Allerdings nimmt die Komplexität auf der Client-Seite stark zu und es kommt bei Caching-Fehlern zu Datenfehlern. Das Orca-Backend unterstützte zum Zeitpunkt der Erstellung dieser Arbeit diesen Constraint (noch) nicht.

Uniform Interface Constraint wird von Fielding als Hauptunterschied zu anderen Netzwerkarchitekturen genannt. Gemeint ist damit, dass eine einheitliche Komponentenschnittstelle benötigt wird. Damit soll eine minimal notwendige (lose) Kopplung der Komponenten erreicht werden.

Als letzten Constraint nennt Fielding den **Layered System** Constraint. Durch Verwendung verschiedener Schichten sollen Abhängigkeiten und auch die Komplexität der Architektur reduziert werden. Nachteilig ist allerdings der hierbei entstehende Overhead, was zur Performancebeeinträchtigung führen kann.

Aktuelle Anforderungen an REST-Architekturen

Das Orca-Backend ist nach aktuellen Anforderungen einer REST-Architektur konzipiert. Deshalb sollen hier auch aktuelle Aspekte und Prinzipien von REST-Architekturen erläutert werden. Weitere Kernprinzipien sind in [25] genannt. Es werden hier speziell nur diese aufgeführt, die das Orca-Backend anwendet.

In [25] wird als ein wichtiges Kernprinzip eine *eindeutige Identifikation* von *Ressourcen* genannt. Unter Ressourcen werden dabei alle Abstraktionen verstanden, die es sich lohnt zu identifizieren. (siehe [25], Seite 12) Gemeint sind damit virtuelle Objekte, wie Dokumente, Bilder sowie auch reale Objekte, die “nach außen“ bekannt gegeben werden (sollen). Eine allgemeine Definition lautet: Eine Ressource ist “eine durch eine gemeinsame ID zusammengehaltene Menge von Repräsentationen“([25], Seite 33). Bezogen auf das Orca-Backend ist eine Entität eine Ressource, die durch (mindestens) eine URI repräsentiert wird.

Hypermedia As The Engine Of Application State (HATEOS) ist ein weiteres Kernprinzip. (siehe [25], Seiten 12-13) Allgemein ist damit die anwendungsübergreifende Verwendung von Verknüpfungen (Links) zwischen Ressourcen zu verstehen. Insbesondere die Steuerung des Applikationsszustandes über Links ist entscheidend für dieses Kernprinzip. Damit ist gemeint, dass der Client durch vom Server erstellte Links von einem Zustand in den nächsten übergeht. Das Orca-Backend ermöglicht es für jede Entität Links anzulegen, die auf andere Entitäten zeigen. Über diese Links sind die Ressourcen abrufbar, auf die dieser Link zeigt.

Ein weiteres Kernprinzip ist die Verwendung von *Standardmethoden*. (siehe [25], Seite 15) Damit ist die korrekte Verwendung und Implementierung des Standardanwendungsprotokolls (HTTP) gemeint, damit eine Kommunikation mit allen Ressourcen möglich ist. Das Orca-Backend gibt als Standardmethoden die Verben GET, POST, PUT, DELETE auf verschiedenen Endpunkten vor. Hierauf wird in Unterabschnitt 3.2.2 genauer eingegangen.

Zuletzt soll noch ein weiterer Aspekt genannt werden, der durch das Orca-Backend erfüllt wird. Gemeint ist die sogenannte *Pageinierung*. (siehe [25], Seite 36) Bei einer Anfrage ans Backend wird nur eine bestimmte Anzahl von Elementen in einer *Page* zurückgegeben. Innerhalb dieser befinden sich neben den abgefragten Elementen auch Links für eine weiterführende Navigation ohne erneute Anfrage. Im Orca-Backend wird dieses Prinzip bei der Abfrage von *EntityPages* oder *LinkPages* verfolgt. Diese enthalten entsprechende Links, die auf vorherige oder nachfolgende Seiten führen, ohne dass der Client eine erneute Anfrage senden muss.

2.4. Zusammenfassung

Im Rahmen dieses Kapitels wurde zunächst das modellgetriebene Entwicklungsparadigma vorgestellt. Entscheidend hierbei ist, dass ein Modell einen bestimmten Fachbereich - somit einen bestimmten Teil einer Software - zu 100 % beschreibt. Dieses Modell wird mit Hilfe einer Domain-Specific Language in die Entwicklung integriert, sodass mit Hilfe eines Generators Code generiert werden kann. Eine Domäne sollte wiederholende und nicht zu komplexe Bereiche beschreiben. Dieser generierte Code ist meistens durch Plattformcode zu ergänzen, um die gewünschten Softwareeigenschaften gewährleisten zu können, denn nur in den seltensten Fällen kann die ganze zu erstellende Software durch ein einziges Modell beschrieben werden.

Anwendungen (Apps) für das mobile Betriebssystem Android sind in der Programmiersprache Java zu schreiben. Es wird eine angepasste JVM (Java Virtual Machine), die sogenannte DVM (Dalvik Virtual Machine), verwendet. Diese unterstützt Prozessorregister und ist durch Hardwarehersteller individuell anpassbar. Geräteportierungen werden dadurch erleichtert. Nebenläufigkeit ist bei der Programmierung von Android-Anwendungen ein sehr wichtiger Aspekt. Langlaufende Prozesse dürfen niemals auf dem Main Thread abgearbeitet werden, denn dies könnte zu Einschränkungen bei der Ausführung von Apps führen (Anwendung blockiert). Mit der Klasse AsyncTask wird ein Werkzeug innerhalb des implementierten SDKs verwendet, das es unkompliziert ermöglicht solche Programmteile auszulagern. Standardmäßig wird ein SQLitedatenbanksystem zur Verfügung gestellt. Es handelt sich um eine relationale Datenbank. Eine Nutzung ist ohne vorherige Installation und Rechteverwaltung von jeder Anwendung heraus möglich.

Das Orca-Backend ist zentraler Bestandteil dieser Arbeit. Die Backend-Schnittstelle wurde für dieses Backend erstellt und optimiert. Es ist ein Backend, das nach der REST-Architektur konzipiert wurde. Grundlegendes zum Aufbau des Backends, sowie eine kurze Erläuterung der Grundlagen von REST-Architekturen schlossen dieses Kapitel ab.

Nachdem Grundlagen vorgestellt wurden, sollen im nächsten Kapitel die Anforderungen für das SDK erörtert und vorgestellt werden.

3. Anforderungen an das SDK

Bei der App-Entwicklung ist ein wichtiges Kriterium die Frage der Datenhaltung. Werden die von dem/der Nutzer/in eingegebenen Daten nur lokal auf dem Gerät gespeichert? Erfolgt eine Anbindung an ein Backend? Ist es eventuell sogar möglich Wahlfreiheit über den Speicherort zu gewähren? Bei vielen Apps wäre dies zumindest angebracht und wünschenswert.

Durch diese Fragestellungen lassen sich die Anforderungen dieser Arbeit ableiten. Das entwickelte Android SDK soll es dem/der App-Entwickler/in ermöglichen, entsprechende Schnittstellen für die lokale Datenhaltung sowie für ein Backend (Orca-Backend) nach eigenem Ermessen verwenden zu können. Dabei soll es drei Möglichkeiten geben: Entweder eine Datenhaltung erfolgt nur im Backend, nur in der lokalen SQLitedatenbank oder es wird eine Kombination aus beiden Schnittstellen gewählt. Welche Anforderungen hierbei zu erfüllen und welche Restriktionen einzuhalten sind, soll dieses Kapitel veranschaulichen.

Anmerkung: In den folgenden Kapiteln wird die im Rahmen dieser Arbeit implementierte Softwarelösung Android SDK oder schlicht nur SDK bezeichnet. Damit inbegriffen ist auch die individuell entwickelte DSL. Ab diesem Kapitel werden die drei Varianten des Android SDKs oftmals als Schnittstellen bezeichnet. Dabei sind keine "echten" Schnittstellen wie Java-Interfaces gemeint. Dieser Begriff ist metaphorisch zu verstehen. Der/die Entwickler/in soll wahlfrei über die Nutzung der Schnittstellen entscheiden können.

3.1. Allgemeine Anforderungen

Ein Domänenmodell soll vor der eigentlichen App-Entwicklung dem/der Entwickler/in die benötigten Entitäten festlegen lassen. Was das Domänenmodell individuell alles leisten muss, wird in den jeweiligen Abschnitten dieses Kapitels erläutert.

Weiterhin soll es ermöglicht werden nur noch mit "echten" Java-Objekten zu programmieren. Der/die Entwickler/in soll von Umwandlungen der Java-Objekte (z.B. ins JSON-Format) nichts mitbekommen. Alle Funktionsweisen des SDKs sollen durch statische Methoden wie auch Instanzmethoden auf den generierten Entitätsklassen aufrufbar sein. Die

3. Anforderungen an das SDK

Methoden sollen dabei nur die minimale Anzahl an Übergabeparametern benötigen.

Egal für welche Variante sich der/die Entwickler/in entscheidet, es sollte gewährleistet sein, dass das fertige Entwicklungswerkzeug selbsterklärend ist. Darunter ist zu verstehen, dass aufrufbare Methoden dokumentiert sind und bestenfalls auch aus dem Methodennamen der eigentliche Zweck erschließbar ist. Implementierungsdetails, die für die vorgesehene Funktion nicht benötigt werden, müssen ebenso verdeckt werden, damit Anwendungsfehler von vornherein bestmöglich ausgeschlossen werden. Ratsam ist eine Schichtenhierarchie für das SDK zu verwenden, die nur die letzte Schicht dem/der Anwender/in zur Verfügung stellt.

3.2. Backend-Schnittstelle

In diesem Abschnitt werden Anforderungen aufgeführt, die das SDK, die DSL und der Generator in Bezug auf die Schnittstelle zum Orca-Backend leisten muss. Abbildung 3.1 veranschaulicht grafisch die benötigten Funktionen.

3.2.1. Grundlegende Spezifikationen

Autorisierung

Eine ganz wichtige Anforderung für das SDK in Bezug auf das Orca-Backend ist die zentrale Festlegung des *Appnamens* und des *Apikeys*. Wie in Abschnitt 2.3.1 bereits erläutert, sind Anfragen nur an das Orca-Backend möglich, wenn in dem entsprechenden Request-Header beide Angaben vorhanden sind. *Appname* und *Apikey* sind für jede zu entwickelnde App einzigartig. Das SDK soll gewährleisten, dass beide Angaben an zentraler Stelle - und am besten nur einmal - vorgenommen werden können und automatisch bei den Requests verwendet werden. Es bietet sich an beide Angaben über das Domänenmodell festzulegen.

Festlegung des EntityType

Die Festlegung des jeweiligen *EntityType* ist für die Funktionsweise des Orca-Backend von entscheidender Bedeutung. Darauf werden Entitäten unterschieden. Im Domänenmodell muss es somit unumgänglich sein, diesen Type (ein eindeutiger Long-Wert) für alle festgelegten Entitäten zwingend festlegen zu müssen. Von Vorteil wäre

3. Anforderungen an das SDK

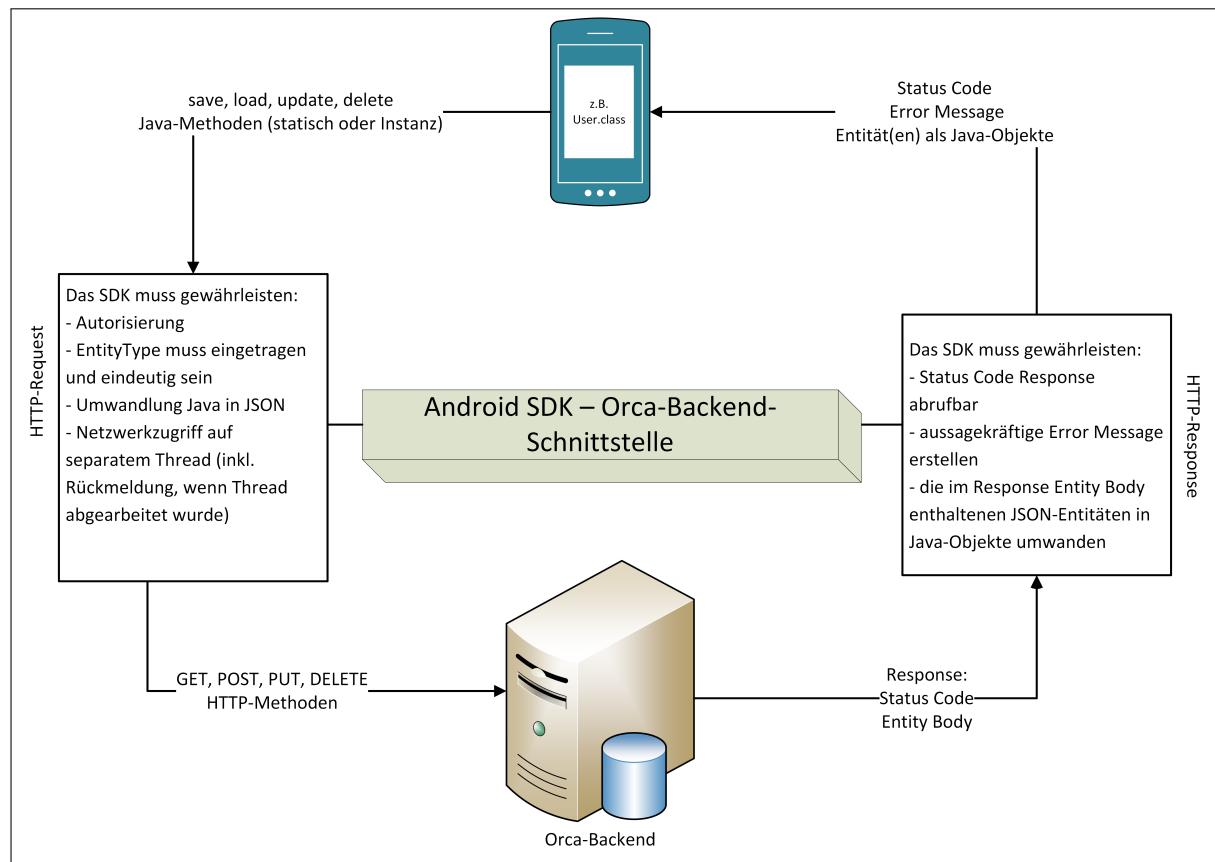


Abbildung 3.1.: Spezifikation der Orca-Backend-Schnittstelle (Quelle: Eigene Darstellung)

3. Anforderungen an das SDK

auch eine Überprüfung der Type-Werte vor der Codegenerierung. Es sollte explizit geprüft werden, ob der Type auch eindeutig ist und - falls Dupletten vorkommen - automatisch ein eindeutiger Wert vergeben wird.

Unterstützung von Links, Images und Pages

Aufgrund der REST-konformen Architektur des Orca-Backends muss das SDK die Erstellung von *Links*, die auf andere Entitäten zeigen, gewährleisten. Alle für eine Entität möglichen *Links* sollen auch über das Domänenmodell definierbar sein. Wichtig hierbei ist, dass neben dem *LinkTyp* auch der Entitäts-Typ der Entität, auf die diese *Links* zeigen sollen, mit festgelegt werden muss. Ebenso sollen *LinkProperties* angegeben werden können.

Das Orca-Backend ermöglicht es jeder Entität Bilder hinzuzufügen. Dementsprechend soll auch das SDK entsprechende Funktionalitäten gewährleisten. *Images* sollen dabei auch über das Domänenmodell festgelegt werden können (Name des Bildes). Die DSL für das Domänenmodell muss die bisher genannten Punkte erfüllen, damit die Konformen des Orca-Backends so gut wie möglich eingehalten werden können.

Auch wurde bereits in vorherigen Kapiteln die Pageinierung erwähnt (siehe Abschnitt 2.3.2). Das SDK soll deshalb *Pages* vom Backend in Java-Collections umwandeln können bzw. Java-Objekte, anhand derer die Daten der *Pages* ausgelesen werden können (*EntityPages* und *LinkPages*, siehe [14]).

Nebenläufigkeit

Das SDK muss einerseits gewährleisten, dass alle Requests nicht auf dem Main Thread ausgeführt werden. (siehe Unterabschnitt 2.2.2) Es kann durchaus längere Zeit dauern, bis eine Response erhalten wird. Um den Main Thread nicht zu blockieren, müssen alle entsprechenden Methoden die Backend-Zugriffe auf einem separaten Thread durchführen. Das SDK muss andererseits auch versichern, dass threadsicher auf die abgefragten Ressourcen zugegriffen werden kann. Hierbei sollte manuell gesteuert werden können, ob eine Rückmeldung bei Beendigung des Threads erfolgen soll oder nicht.

Fehlerhandhabung

Bei Netzwerkzugriffen gibt es viele Fehlerquellen. Das Umwandeln eines Java-Objektes in ein JSON-Objekt könnte so eine Fehlerquelle darstellen. Es würde nun wenig Sinn machen, wenn das SDK alle Fehler unbehandelt an den/die Entwickler/in zurückgibt

3. Anforderungen an das SDK

(per Exception). Fehler sollen über einen einheitlichen Fehlercode (z.B. -1) abprüfbar sein. Dadurch kann eine einfach `if-else`-Abfrage klären, ob ein Fehler aufgetreten ist oder nicht. Ebenso wäre von Vorteil, wenn eine Fehlermeldung als String beschreibt, welche/r Fehler aufgetreten sind/ist. Exceptions sollen dabei abgefangen werden.

3.2.2. HTTP-Standardmethoden

Aufgrund der REST-Konformität muss eine Anzahl von Operationen bereitgestellt werden, die für alle Ressourcen (Entitäten) gleichermaßen gültig sind. (siehe [25], Seite 51) Das Orca-Backend bietet hierfür verschiedenste Endpunkte an, auf denen bis zu vier Methoden (Verben) ausgeführt werden können. Mit Methoden sind hier im speziellen HTTP-Verben gemeint. HTTP bietet mit seinen Verben genau diese bestimmte Anzahl an Operationen, die benötigt werden. Anbei werden die vier Verben mit ihren Aufgabenbereichen und der Bedeutung für das SDK näher vorgestellt. Bei allen Verben soll es für den/die Entwickler/in möglich sein, den Status Code der Response abfragen zu können.

Anmerkung: Es werden hier nicht die individuellen Endpunkte des Orca-Backends aufgeführt. Eine entsprechende ausführliche Beschreibung der vorhandenen Endpunkte ist unter [15] einsehbar.

GET

“Gemäß Spezifikation dient GET dazu, die Informationen, die durch die URI identifiziert werden, in Form einer Entity (...) abzuholen“ ([25], Seite 51). GET ist die wichtigste HTTP-Methode. (siehe [25], Seite 51) Das **Laden** einer Entität aus dem Backend ist über dieses Verb abzubilden. Wichtig hierbei ist, dass ein GET-Request als *sicher* und *idempotent* gilt. *Sicher* bedeutet, mit dieser Anfrage werden keine Verpflichtungen eingegangen. (siehe [25], Seite 14) Diese Methode ruft somit nur Informationen ab und verursacht keine sonstigen Effekte. Mit *idempotent* ist gemeint, dass das mehrfache Absenden der gleichen GET-Request sich nicht anders auswirkt als ein einzelner GET-Request. Bei einer erfolgreichen GET-Request ohne Fehler wird ein Status Code mit dem Wert 200 in der Response zurückgegeben, der Response-Body soll hierbei nicht leer sein (siehe [12], Seite 28). Das Orca-Backend gibt auch bei einer Suchanfrage, zu der keine Entitäten gefunden wurden, den Status Code 200 und eine leere EntityPage im JSON-Format zurück.

GET-Requests sind für das Android SDK ein Muss, repräsentieren diese einen Befehl für das *Laden* von Entitäten.

3. Anforderungen an das SDK

POST

POST ist zum **Anlegen** von neuen Ressourcen gedacht. (siehe [25], Seiten 54-55) Dieser Befehl soll auch verwendet werden, wenn keine der anderen Methoden passend sind. Hierbei bestimmt der Server die dazugehörige URI der zu erstellenden Ressource. Die URI der Request (diese wird vom Client gesetzt) gibt hierbei nicht die Ressource an, sondern die URI der für das Anlegen zuständigen (Listen-)Ressource. Beim Orca-Backend erfolgt das Anlegen einer neuen Ressource über den Endpunkt `/entities`. Ein POST-Request erfolgt demnach auf einem allgemeinen Endpunkt. Als Status Code der Response bei einer erfolgreichen Anlegung ist der Wert 201 vorgesehen (siehe [12], Seite 28).

Da das SDK das Anlegen (*Speichern*) neuer Entitäten (Ressourcen) gewährleisten muss, ist eine entsprechende Implementierung notwendig.

PUT

Mit PUT ist eine bestehende Ressource (Entität) zu **aktualisieren**. Ist eine Ressource noch nicht vorhanden, ist diese durch einen PUT zu **erzeugen**. (siehe [25], Seite 54) Auch ein PUT ist *idempotent*. Die URI ist hierbei Ziel des Request und wird vom Client festgelegt. Ein Put wirkt sich direkt auf die Ressource aus und ist sozusagen der inverse Befehl zu einem GET. Innerhalb des Request Body müssen die Informationen mitgegeben werden, mit der die Ressource aktualisiert bzw. erstellt werden soll.

Das SDK hat in diesem Zusammenhang z.B. dafür zu sorgen, dass bei einem PUT auf eine Entität im Request-Body eine aktualisierte Repräsentation der Entity im JSON-Format enthalten ist (*Aktualisieren*). Bei einer fehlerfreien Request soll der Response Status Code 204 betragen (siehe [12], Seite 29).

DELETE

DELETE ist das Verb für das **Löschen** von Ressourcen. (siehe [25], Seite 55) Die URI, auf die das DELETE angewendet werden soll, ist vom Client im Request festzulegen. Auch diese Methode ist *idempotent*. Bei erfolgreichem Löschvorgang ist in der Response als Status Code der Wert 204 zu erwarten (siehe [12], Seite 29).

Das Löschen von Entitäten muss über das SDK gewährleistet werden. Auch für dieses Verb ist über die entsprechenden Endpunkte eine Java-Methode durch das Android SDK zur Verfügung zu stellen.

3.3. Lokale Datenbank-Schnittstelle

In diesem Kapitel wird auf die Anforderungen der Datenbank-Schnittstelle Bezug genommen. Es erfolgt analog zum vorherigen Kapitel eine Erörterung der allgemeinen Spezifikationen und die für diese Schnittstelle benötigten Methoden.

3.3.1. Überblick

Bevor auf die Methoden, die von dieser Schnittstelle gewährleistet werden müssen, eingangen wird, soll zunächst die allgemeine Funktion dieser Schnittstelle erläutert werden. Die SQLitedatenbank-Schnittstelle soll es ermöglichen, über das Domänenmodell definierte Entitäten lokal mit der SQLitedatenbank zu verwalten. Es soll somit das gleiche Domänenmodell wie bei der Orca-Schnittstelle verwendet werden können. Der/die Entwickler/in soll auch bei dieser Schnittstelle nur anhand der Java-Objekte entsprechende Funktionalitäten nutzen können. Datenbankspezifische Objekte, wie z.B. `Cursor`, sollen jederzeit verborgen bleiben. Zum besseren Verständnis soll für diese Schnittstelle Abbildung 3.2 herangezogen werden. Prinzipiell ist diese Schnittstelle als einfache Lösung einer lokalen Datenhaltung für Entitäten gedacht. Auch soll durch diese Schnittstelle eine lokale “Zwischenspeicherfunktion“ für Entitäten ermöglicht werden.

Der/die Nutzer/in des SDKs soll wie bei der Orca-Schnittstelle Java-Methoden aufrufen, die die entsprechenden Datenbankbefehle ausführen. Doch ein gravierender Unterschied zur Orca-Schnittstelle ist, dass in eine relationale Datenbank (siehe Abschnitt 2.2.3) die selben Entitäten überführt werden sollen, denn das Domänenmodell soll für die Nutzung dieser Schnittstelle nicht verändert werden müssen. Das Orca-Backend benötigt aufgrund dessen Generizität (siehe Abschnitt 2.3.1) kein Tabellschema, dass vor dem erstmaligen Speichern erstellt werden muss. Bei der SQLitedatenbank ist dies zwingend notwendig. Beide Schnittstellen sollen zueinander in Bezug auf die Entitäten kompatibel sein, d.h. eine aus dem Orca-Backend geladene Entität soll ohne weiteres per SQLitedatenbank-Schnittstelle lokal gespeichert werden können und umgekehrt.

3.3.2. Datenbankmethoden

Um zu definieren, welche Datenbankmethoden durch das SDK abgedeckt werden müssen, ist es nötig die wichtigsten SQL(ite)-Befehle zu kennen. Allgemein können SQL(ite)-Befehle in verschiedene Kategorien eingeteilt werden. (siehe [1], Seite 63, bezieht sich auf alle diesem Abschnitt zugehörigen Absätze) Drei Kategorien werden wie folgt definiert:

Data Definition Language (DDL) wird der erste Bereich genannt. Hierunter fallen

3. Anforderungen an das SDK

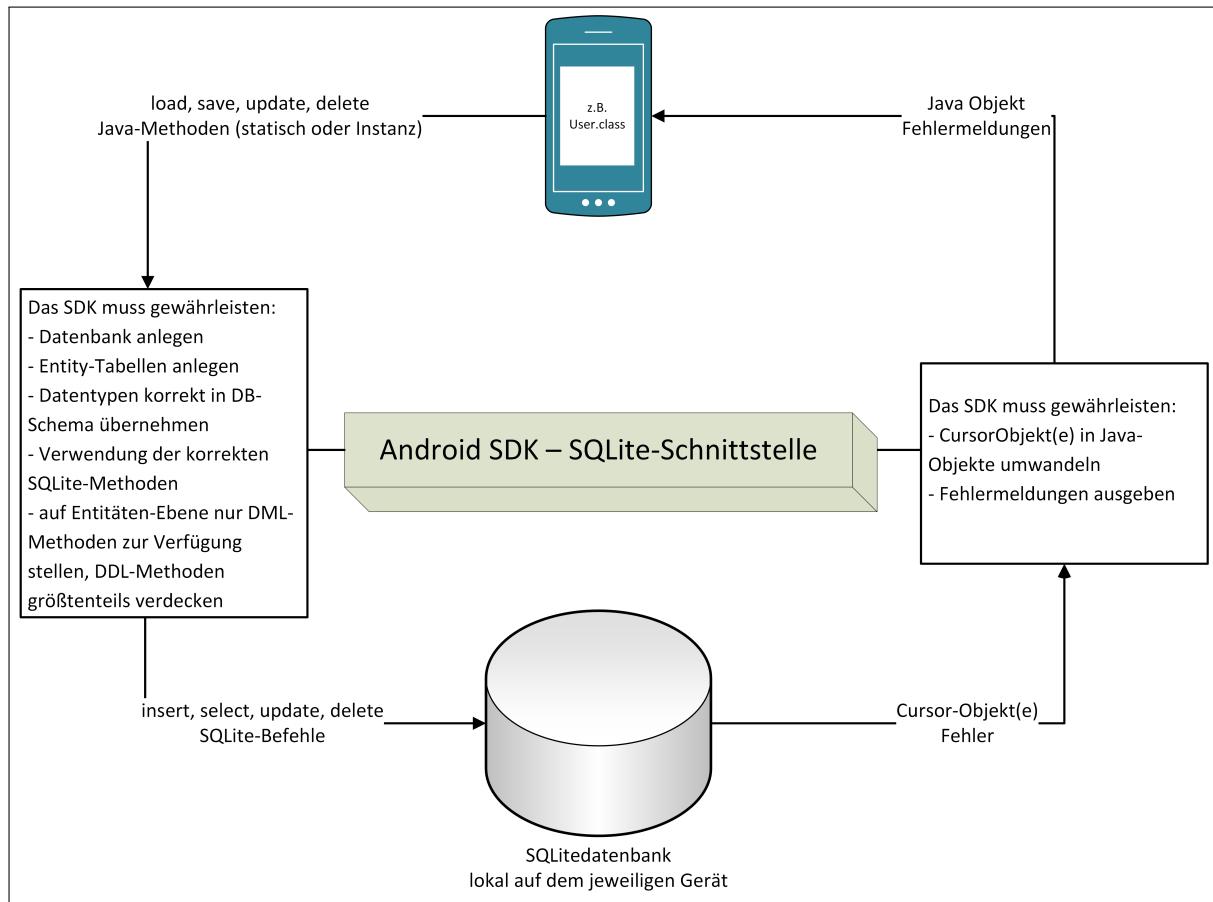


Abbildung 3.2.: Spezifikation der SQLitedatenbank-Schnittstelle (Quelle: Eigene Darstellung)

3. Anforderungen an das SDK

alle Befehle, die zum Anlegen, Ändern und Löschen von *Datenbanken, Tabellen und deren Strukturen* benötigt werden. Ein Beispiel wäre ein CREATE TABLE Befehl.

Data Manipulation Language (DML) ist eine weitere Kategorie. Hierunter werden Befehle zum Einfügen, Ändern, Löschen und Auslesen von *Daten aus den Tabellen* zusammengefasst. Ein INSERT-Befehl gehört demnach zu dieser Kategorie.

Data Control Language (DCL) fasst Befehlssätze zum Administrieren von Datenbanken zusammen. Hier ist ein GRANT-Befehl ein entsprechendes Beispiel.

In Unterabschnitt 3.3.1 wurde bereits genannt, dass mit dieser Schnittstelle eine “Zwischenspeicherfunktion“ für Entitäten ermöglicht werden soll. Damit sollen alle Datenbankfunktionen im Hintergrund ablaufen. Der/die Entwickler/in sollen für die Verwendung lediglich Methoden auf den entsprechenden Java-Entitäten ausführen. Grundsätzlich werden analog zur Orca-Schnittstelle Methoden zum Speichern, Laden, Aktualisieren und Löschen von Tabelleneinträgen benötigt. Auch macht eine Reset-Funktion zum kompletten Zurücksetzen der Datenbank Sinn. Demnach werden nach den obigen Definitionen Befehlssätze aus den Bereichen DDL und DML benötigt. DCL-Befehlssätze müssen nicht abgedeckt werden.

Welche SQLite-Befehle aus welchem Bereich genau benötigt werden, wird folgend erläutert.

DDL - CREATE

Im Bereich DDL wird ein CREATE-Befehl z.B. zum Erstellen von Tabellen benötigt. Bei einem Vergleich mit der Orca-Schnittstelle ist auffällig, dass dort ein entsprechender Befehl nicht notwendig ist. Dies liegt an der Generizität des Orca-Backends (siehe Generizität). Für den/die Nutzer/in des SDKs wäre es leicht verwirrend, wenn bei Nutzung der SQLitedatenbank-Schnittstelle zuerst manuell eine Tabelle erstellt werden müsste.

Dieser Befehl soll deshalb ausschließlich durch das SDK intern benutzt werden, um z.B. eine Tabelle anhand eines Schemas zu erstellen. Hier ist wichtig, dass das SDK automatisch diesen Befehl verwendet und intern kapselt.

DDL - DROP

Ein DROP-Befehl kann zum Löschen einzelner Tabellen oder der gesamten Datenbank verwendet werden. Ein Pendant hierzu ist bei der Orca-Schnittstelle aus gutem Grunde nicht vorhanden, denn es wäre sonst der gesamte Datenbestand des Backends lösbar.

3. Anforderungen an das SDK

Bei der Orca-Schnittstelle macht dieser Befehl somit keinen Sinn. Auf lokaler Ebene ist dies anders. Gerade weil die SQLitedatenbank als “Zwischenspeicher“ verwendet werden können soll, ist es sehr sinnvoll Methoden anzubieten, die die gesamte Tabelle eines bestimmten Entitätstyps bzw. die ganze Datenbank mit allen bis dahin enthaltenen Tabellen löscht.

DML - INSERT

INSERT dient zum Anlegen einzelner oder mehrerer Zeilen in Datenbanktabellen. Mit Zeile ist ein Datensatz gemeint. Eine zu speichernde Entität wäre über einen INSERT-Befehl in eine Tabelle einzufügen. Dieser Befehl ist demnach mit dem POST oder PUT bei der Orca-Schnittstelle vergleichbar. Eine entsprechende Methode des SDKs könnte saveToDatabase() lauten. Diesen Befehl muss das SDK abdecken können.

DML - SELECT

Mit dem SELECT-Befehl lassen sich Datensätze aus der Datenbank abfragen. Entscheidend hierbei ist die WHERE-Klausel. Hierüber können Teilmengen abgefragt werden. Dieser Befehl ist mit der GET-Methode der Orca-Backend-Schnittstelle vergleichbar. Zum Laden von lokal gespeicherten Entitäten muss dieser Befehl in entsprechender Weise in das SDK involviert werden. Denkbar wäre eine Methode, der nur der Teil der WHERE-Klausel zu übergeben ist.

Ein weiterer wichtiger Punkt ist, dass die Datenbank die gefundenen Datensätze als CURSOR-Objekt zurückgibt. Das SDK muss gewährleisten, dass aus diesem Ergebnis ein Java-Objekt erstellt und zurückgegeben wird.

DML - UPDATE

Zum Aktualisieren von Datensätzen wird ein UPDATE-Befehl benötigt. Die Orca-Schnittstelle verwendet hierfür die PUT-Methode. Es wäre möglich eine WHERE-Klausel mitzugeben. Die Fehlergefahr würde dadurch allerdings erheblich steigen. Das SDK soll demnach nur eine Methode anbieten, die nur die Entität aktualisiert (sozusagen überschreibt).

3. Anforderungen an das SDK

DML - DELETE

DELETE wird zum Löschen bestimmter Datensätze benötigt. Im Prinzip ist dieser Befehl gleich zum UPDATE-Befehl, es wäre auch hier möglich eine WHERE-Klausel zu definieren. Auch hier macht nur eine SDK-Methode Sinn, die direkt auf dem Objekt ausgeführt wird (Instanzmethode). Die Instanz kann nach der Ausführung durchaus weiterhin vorhanden sein, nur der zugehörige Datenbankeintrag soll gelöscht werden.

3.4. Kombination von Backend und Datenbank

Grundsätzlich wäre nun anzunehmen, dass mit der Realisierung der vorhergehenden Schnittstellen eine durchaus ausreichend vereinfachte Handhabung der Datenhaltung von Android-Apps gesorgt wird. Doch betrachtet man aktuelle Apps, würden Entwickler/innen mit beiden Schnittstellen schnell an Grenzen stoßen. Oftmals wird ein automatisches Zusammenspiel von lokaler und zentraler Datenhaltung benötigt.

Gründe für eine kombinierte Schnittstelle

Bei der Appentwicklung stellt sich somit noch ein weiterer Sachverhalt, der nicht unberachtet bleiben soll. Durch eine Kombination der Orca- und SQLitedatenbank-Schnittstelle könnte z.B. eine Methode realisiert werden, die Entitäten aus dem Orca-Backend lädt und automatisch in die SQLitedatenbank speichert. Es wäre somit möglich, Entitäten lokal über getter- und setter-Methoden zu bearbeiten. Nach der Bearbeitung wäre ein Update der Entitäten im Backend erforderlich, wofür ebenfalls eine entsprechende Methode durch das SDK angeboten werden soll.

Dies hätte einige Vorteile:

- Entitäten könnten konsistent im Backend und der Datenbank gehalten werden - nur durch Aufruf einzelner Java-Methoden.
- Die Netzlast könnte auf ein Minimum reduziert werden, da weniger Backendzugriffe benötigt werden. Entitäten können nämlich lokal per setter-Methoden bearbeitet werden. Nach Abschluss der lokalen Bearbeitung kann ein Update im Backend erfolgen. Es ist somit nur zu Beginn und nach Beendigung der Bearbeitung ein Backendzugriff notwendig.
- Durch die reduzierte Netzlast würde sich auch das allgemeine Laufzeitverhalten der Anwendung verbessern, da lokale Zugriffe auf die Datenbank wesentlich schneller

3. Anforderungen an das SDK

durchgeführt werden können.

- Dem/der Entwickler/in wird eine eigene Implementierung einer solchen Kombi-Schnittstelle abgenommen.
- Die SQLite-Schnittstelle würde seiner Rolle als “Zwischenspeicher“ entsprechend gerecht werden (können).

Auch bei dieser Schnittstelle müssen Methoden zum Laden, Speichern, Aktualisieren und Löschen von Entitäten vorhanden sein. Entscheidend ist allerdings der Workflow, der durchlaufen werden soll. Aus folgenden Gründen macht es Sinn, immer (!) zuerst das Backend anzusteuern:

- Über das Backend wird für jede Entität eine eindeutige ID vergeben. Da die lokale Datenbank zurückgesetzt werden kann, würde es zu Fehlern kommen, denn die lokale Datenbank würde nach einem Reset mit der ID “1“ beginnen.
- Darüber hinaus legt das Backend die Self-URI fest. Würde erst die lokale Datenbank die Entität speichern, würde dort eine Entität ohne Self-URI gespeichert werden. Somit müsste nach erfolgreichem Abgleich mit dem Backend nochmals eine Aktualisierung der lokalen Datenbank erfolgen.
- Ebenso können nur über das Orca-Backend *Links* und *Images* angelegt oder aktualisiert werden.

In Abbildung 3.3 werden die Spezifikationen der Kombinations-Schnittstelle grafisch vereinfacht zusammengefasst.

3. Anforderungen an das SDK

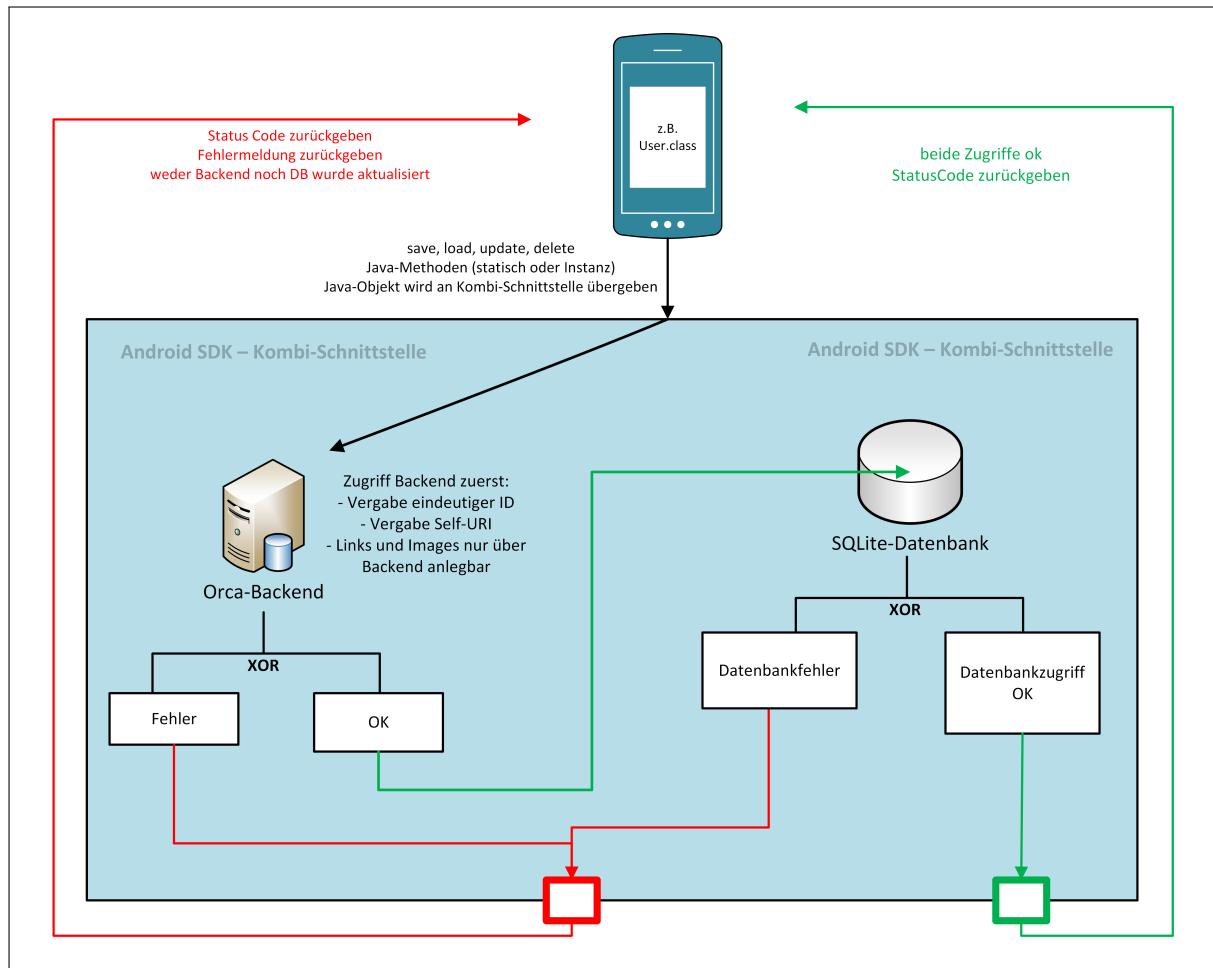


Abbildung 3.3.: Spezifikation der kombinierten Schnittstelle (Quelle: Eigene Darstellung)

3.5. Zusammenfassung

In diesem Kapitel wurden die Anforderungen für das zu implementierende Android SDK spezifiziert.

Das Domänenmodell soll vorrangig auf die in diesem Kapitel erwähnten Konformen des Orca-Backends abzielen: Es soll ermöglichen, Appname und Apikey sowie alle benötigten Entitäten und deren individuellen EntityType-Werte festzulegen. Daraus sollen entsprechende Java-Klassen generiert werden. Die generierten Klassen gewährleisten eine korrekte und intuitive Verwendung des SDKs in der Form, dass rein auf Java-Objekten programmiert werden kann.

Das SDK soll drei abgegrenzte Schnittstellen anbieten, die es ermöglichen, Entitäten für Android-Anwendungen entweder in einem Backend (Orca-Backend), in der auf dem jeweiligen Gerät standardmäßig enthaltenen SQLitedatenbank sowie einer automatisierten Kombination der beiden vorherig genannten Schnittstellen zu verwalten. Die Backend- und SQLitedatenbank-Schnittstelle sollen zueinander kompatibel sein. Insbesondere das Domänenmodell soll alle drei Varianten ohne Anpassungen unterstützen.

Alle drei Schnittstellen müssen einheitlich das Laden, Speichern, Aktualisieren und Löschen von Entitäten ermöglichen.

Das folgende Kapitel wird die Umsetzung und Implementierung des SDKs erläutern. Insbesondere wird darauf eingegangen, ob die in diesem Kapitel festgelegten Spezifikationen erfüllt werden.

4. Modellgetriebene Entwicklung des Android SDKs

Nachdem im vorherigen Kapitel festgelegt wurde, was das Android SDK alles zu leisten hat, wird nun auf die Umsetzung und Implementierung der geforderten Spezifikationen eingegangen. Dabei wird das Android SDK in zwei Bereiche aufgeteilt: Codegenerierung und Plattformcode.

Codegenerierung wird die entwickelte DSL vorstellen, welche mit Xtext umgesetzt wurde. Ebenso wird unter diesem Punkt der Codegenerator veranschaulicht. Hier wird insbesondere auf die in Kapitel 3 definierten Restriktionen eingegangen.

Der zweite Bereich befasst sich mit dem Plattformcode des Android SDKs. Mit diesem Begriff wird der Code angesprochen, der das “eigentliche” SDK darstellt. Darunter ist ein Framework bestehend aus verschiedenen (größtenteils generischen) Klassen zu sehen, die den Orca-Backend-Zugriff und SQLitedatenbankzugriff bewerkstelligen.

Im Anschluss wird darauf eingegangen, wie generierter Code und Plattformcode ineinander greifen und beide Komponenten zu “einem” Android SDK werden lassen.

Ein kurzer Überblick über die regelmäßig durchgeführte Qualitätssicherung ist ebenso Bestandteil dieses Kapitels.

Es soll an dieser Stelle bereits vorweggenommen werden, dass die geforderten Spezifikationen fast vollständig eingehalten werden. Die in Abbildung 3.1 und Abbildung 3.2 aufgeführten Spezifikationen werden erfüllt. Für diese Schnittstellen werden die für die Erfüllung verantwortlichen und entscheidenden Konzepte der Implementierung betrachtet. Bei der Kombinationsschnittstelle mussten gewisse Kompromisse eingegangen werden, die in diesem Kapitel in Abschnitt 4.7 erläutert werden.

4.1. Codegenerierung

4.1.1. DSL mit Xtext

Die DSL wurde mit dem primären Ziel einer vollständigen Abdeckung der Orca-Backend-Spezifikationen entwickelt. In Anhang A ist der gesamte Quellcode der selbst entwickelten DSL abgedruckt, auf die sich die folgenden Ausführungen beziehen.

Der Einstiegspunkt ist in Zeile 5 und Zeile 6. Dort wird die erste Parserregel definiert. Zuerst ist demnach die Parserregel in Zeile 8 zu erfüllen. Die in einfachen Anführungszeichen und roter Schrift markierten Zeichen veranschaulichen die einzuhaltende Syntax. In diesen Zeilen wird demnach festgelegt, dass zuallererst die AppConstants *Appname* und *Apikey* einzutragen sind. Entscheidend ist auch, dass in Zeile 6 festgelegt wird, dass genau eine (oder keine) Parserregel AppConstants erfüllt werden muss. Dies ist durch das “=” und “?” gekennzeichnet. Ein Umgehen dieser Festlegung ist nicht möglich. Eine wichtige Spezifikation in Bezug auf Konformität gegenüber dem Orca-Backend wird somit schon gewährleistet.

In Zeile 13 wird festgelegt, dass als nächstes ein (Java-)Package festzulegen ist. Der/-die Entwickler/in muss als nächstes somit ein Java-Package definieren, zu welchem die danach folgenden Entitäten gehören sollen. Es wäre auch möglich gewesen, ein Default-Package festzulegen. Aus Gründen der Übersichtlichkeit sollen aber Package-Strukturen angelegt werden können. Durch “+=“ und dem “+“ nach Packages ist definiert, dass mindestens ein Package anzugeben ist (siehe [27]).

Der Body einer Packages-Regel (Zeile 16-20) muss mindestens ein AndroidEntity beinhalten. Ein AndroidEntity (Zeile 26-31) ist durch einen Namen zu definieren. Dieser steht für den Java-Klassennamen. In Zeile 28 ist als erste Angabe in einem AndroidEntity-Body der TYPE zu setzen. Es muss eine ganzzahlige Zahl hierfür vergeben werden. Ein Umgehen dieser Angabe ist nicht möglich. Hierdurch wird eine weitere wichtige Vorgabe für das Orca-Backend eingehalten. Anschließend ist mindestens ein AndroidAttribute festzulegen.

Diese Parserregel (Zeilen 33-42) legt die erlaubten Datentypen der einzelnen Attribute einer Entität fest. Auch hier werden nur die Datentypen erlaubt, die das Orca-Backend verarbeiten kann. In Zeilen 40 und 41 sind als “Datentyp“ auch ein image und ein link aufgeführt. Hiermit werden die entsprechenden Funktionalitäten des Orca-Backends unterstützt.

In Zeile 41 ist festgelegt, dass für einen link ein Name und ein objectType anzugeben ist. Der Name spiegelt hierbei den für das Orca-Backend erforderlichen LinkType dar. Da ein Link immer auf eine Entität zeigen muss, ist ein objectType anzugeben. Dabei

4. Modellgetriebene Entwicklung des Android SDKs

muss dieser vom Typ einer `AndroidEntity` sein, das innerhalb des gleichen Package definiert wurde.

In den Zeilen 46-48 werden Datentypen für `LinkProperties` festgelegt (siehe hierzu Abschnitt 2.3.1). Auch hierbei handelt es sich um Datentypen, die kompatibel zum Orca-Backend sind.

Was genau die Festlegungen für `image` und `link` im generierten Code bewirken, wird in Abschnitt 4.3 beschrieben.

Domänenmodell

Zum besseren Verständnis der eben vorgestellten DSL wird hier das entsprechende Domänenmodell näher betrachtet, das mit gezeigter DSL realisiert werden kann.

Listing 4.1: Beispiel Domänenmodell (Quelle: Eigener Quellcode)

```
1 appconstants {
2     appname = "meier";
3     apikey = "lasljdkaflfasldjfajdfajsd23424";
4 }
5
6 package de.test.orcasdk{
7     entity person{
8         TYPE = 100;
9         string name;
10        string vorname;
11        date geburtstag;
12        double[] standort;
13        link kenntJemand person{
14            string nachricht;
15        }
16        image profilbild;
17    }
18 }
```

Anhand dieses Beispiels wird die einzuhaltende Syntax verdeutlicht. Treten hierbei Fehler auf bzw. wird die korrekte Form nicht eingehalten, würden Compilerfehler angezeigt. Ebenso werden alle vergebenen Namen (für `package`, `entity` und alle `android Attribute`) auf Eindeutigkeit geprüft. Bei doppelter Vergabe von Namen würde ebenfalls ein Fehler angezeigt werden. Diese Eigenschaft des Domänenmodells wurde anhand eines in Xtext verfügbaren `DefaultValidator` umgesetzt. Nähere Erläuterungen hierzu sind aus [3] (Seite 72) entnehmbar.

4.1.2. Generator mit Xtend

Der Generator wurde so konstruiert, dass anhand des `fsa`-Objektes alle Parserregeln ausgelesen werden. Das Auslesen erfolgt durch entsprechende `for`-Schleifen. Nachdem die entsprechenden jeweils benötigten Werte ermittelt wurden, erfolgt eine Generierung der Java-Klasse. In Listing 4.2 ist der Einstiegspunkt des Generators (`doGenerate`-Methode) zu sehen. Diese ruft eine weitere Methode auf, in der nach `appconstants` gefiltert wird. Innerhalb der Schleife erfolgen weitere Methodenaufrufe, die zuerst nach `Packages` filtern und auch die Generierung von Code veranlassen.

Listing 4.2: Ausschnitt Xtend-Generator (Quelle: Eigene Quellcode)

```

1 ...
2 override void doGenerate(Resource resource, IFileSystemAccess fsa) {
3
4     loopOverAppConstants(resource, fsa);
5 }
6
7 def loopOverAppConstants(Resource resource, IFileSystemAccess fsa) {
8
9     for(appconstants:
10        resource.allContents.toIterable.filter(AppConstants)) {
11         loopOverPackages(resource, fsa)
12         generateAppConstantsClass(fsa, appconstants)
13         generateBackendPersistenceWrapper(fsa)
14         initializeEntityTypeCheckVariables()
15     }
16 ...
17 ...
18 ...
19 def generateAppConstantsClass(IFileSystemAccess fsa, AppConstants
20     appconstants) {
21     fsa.generateFile("de/fhws/sdk/orca"+"/"+AppConstants+".java",
22         CreateAppConstants.appconstantscompile(
23             appconstants.valueappname.
24             toString.replace("[", "").replace("]", ""),
25             appconstants.valueapikey.toString.replace("[", "").
26             replace("]", "")));
27 }
28 ...

```

In Zeile 14 erfolgt der Aufruf der `generateFile`-Methode. Dort wird die Java-Klasse `de.fhws.sdk.orca.AppConstants.java` erstellt. In diesem Fall ist das zugehörige Package fest vorgegeben. Bei den Entitätsklassen würde hier der individuelle Packagename übernommen werden. Das für die Generierung von `AppConstants` benötigte

4. Modellgetriebene Entwicklung des Android SDKs

Xtend-Template ist in die Klasse `CreateAppConstants` in deren statische Methode `appconstantscompile` ausgelagert. Dort wird auf das benötigte Template zugegriffen. In Listing 4.3 ist der entsprechende Xtend-Code abgedruckt.

Listing 4.3: AppConstants-Xtend-Template (Quelle: Eigener Quellcode)

```
1 def static appconstantscompile(String appName, String apiKey) {
2 /**
3 package de.fhws.sdk.orca;
4
5 public class AppConstants {
6
7     public static final String APPNAME = "<<appName>>";
8     public static final String APIKEY = "<<apiKey>>";
9
10 }
11 /**
12 }
```

Dieser Methode werden die aus dem `fsa`-Objekt ermittelten Werte für `appname` und `apikey` übergeben. Im Template werden diese (Zeilen 7-8) an die entsprechenden Stellen in der eigentlichen Java-Klasse “gesetzt“.

Grundsätzlich werden nach obigem Ablauf alle weiteren Parserregeln durchlaufen und eine entsprechende Codegenerierung wird angestoßen. Von weiteren Ausführungen hierzu wird deshalb abgesehen.

Auf Zeile 13 in Listing 4.2 soll hier noch explizit hingewiesen werden. Dort wird eine Prüfung und ein Abgleich aller im Domänenmodell vergebener `TYPE`-Werte durchgeführt. Diese müssen für jede Entität eindeutig sein. Damit es hier nicht zur mehrfachen Vergabe des gleichen Wertes kommt, wird in dieser Methode für alle Entitäten (auch Package-übergreifend) ein Abgleich durchgeführt. Falls tatsächlich ein Wert schon vergeben wurde, wird der gesetzte Wert solange um eins erhöht, bis dieser eindeutig ist.

Wichtig zu erwähnen ist die gewählte *Package-Struktur*. Eine Ausgabe aller generierten Klassen erfolgt immer in den `src-gen`-Ordner. Innerhalb dieses Ordners werden die entsprechenden Packages angelegt, die im Domänenmodell definiert wurden, wo auch die festgelegten Entitäten gespeichert werden. Hier wird auch eine Klasse `EntityTypeConstants` angelegt, worüber die `EntityType`-Werte der generierten Entitäten abgefragt werden können. Alle Klassen, die einheitlich für alle generierten Entitätsklassen gültig sind, werden in das Package `de.fhws.sdk.orca` gespeichert. Die Klasse `AppConstants` gehört z.B. hierzu.

Um weitere Besonderheiten der Codegenerierung verständlich anzusprechen, ist es notwendig zu erläutern, wie der generierte Code mit dem Plattformcode verknüpft ist.

4.2. Zusammenspiel von generiertem Code und Plattformcode

Innerhalb des Plattformcodes wird eine zentrale Java-Klasse Entity vorgegeben. Von dieser Klasse erben alle generierten Entitätsklassen. Dabei ist entscheidend, dass Entity ein Java-Abbild der JSON-Entity des Orca-Backends (siehe hierzu Abschnitt 2.3.1) darstellt. In Abbildung 4.1 ist ein entsprechendes UML-Klassendiagramm (Aufbau des Diagramms nach [10]) abgebildet.

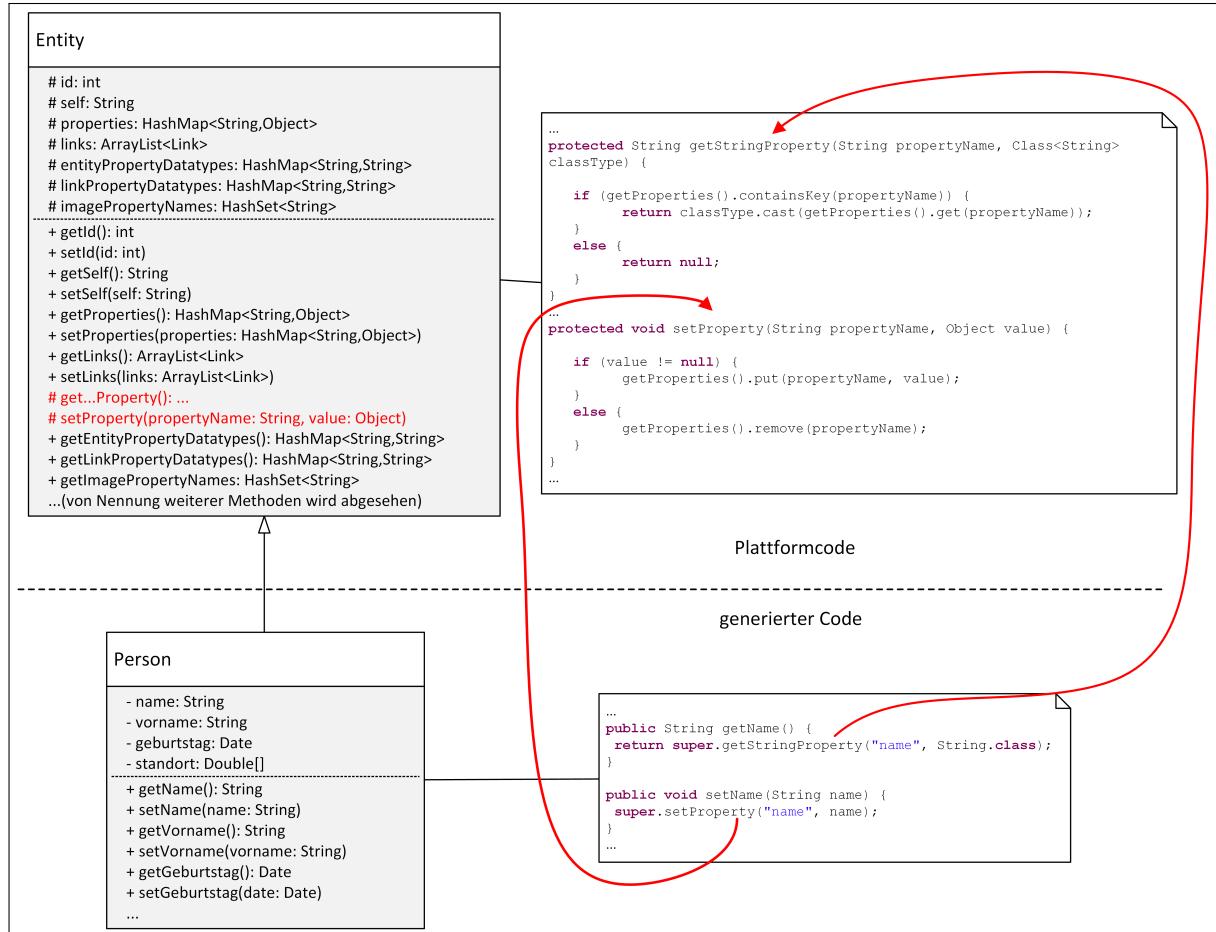


Abbildung 4.1.: UML-Klassendiagramm Entity-Workflow (Quelle: Eigene Darstellung)

Alle Attribute einer Entität werden in einer `HashMap<String, Object>` der Entity-Klasse abgebildet (`properties`). Die Attributnamen sind hierbei die entsprechenden Key-Strings, die zugehörigen Attributwerte sind Value-Objekte. Da alle Value-Objekte vom Typ `Object` sein müssen, können hier verschiedenste Datentypen hinterlegt werden. Alle Attributwerte werden ausschließlich in der Oberklasse in dieser `HashMap` gehalten.

4. Modellgetriebene Entwicklung des Android SDKs

Bei der Generierung der Entitätsklassen werden die Getter- und Setter-Methoden entsprechend nach den im Domänenmodell definierten Namen und Datentypen konstruiert. Dies ist in Abbildung 4.1 ebenso veranschaulicht. In der Entity-Klasse gibt es Getter-Methoden für alle konformen Datentypen. Diese Methoden haben das Namensformat `get{spezifischerDatentyp}Property` (ohne geschweifte Klammern).

Die eigentlichen Getter- und Setter-Methoden der Unterklassie kapseln den Aufruf dieser Entity-Getter-Setter-Methoden. Hierüber werden alle Attributwerte aus der `properties`-HashMap gelesen und gesetzt. Da innerhalb der `properties`-HashMap alle Value-Objekte vom Typ `Object` sind, muss beim Lesen ein entsprechender Cast auf den eigentlichen Property-Datentyp erfolgen. Für diesen Zweck müssen diese datentypspezifischen Getter-Methoden in der Entity-Klasse implementiert und vererbt werden. Die eigentlichen Attribute, die in der Unterklassie durch den Generator angelegt werden (bezogen auf die in Abbildung 4.1 gezeigte Klasse `Person` z.B. das Attribut `name`), sind nur für die Umwandlung mit Genson notwendig. Die Attributwerte werden ausschließlich in genannter HashMap verwaltet. Aus Sicht des/der Anwenders/Anwenderin ist hierbei kein Unterschied zur gewöhnlichen Implementierung von Getter-Setter-Methoden bemerkbar.

Diese Herangehensweise wurde aus einem einfachen Grund gewählt. Eine Vielzahl zentraler Klassen des Plattformcodes lassen sich dadurch generisch implementieren. All jene Klassen des Plattformcodes erwarten dabei als generischen Parameter immer eine Klasse, die von Entity ableitet. Zentrale Klassen können dadurch auf der Plattformseite generisch implementiert werden, welche ansonsten für jede Entität generiert werden müssten. Dadurch soll der zu generierende Code minimal gehalten werden.

Durch Kapselung aller Attribute in einer `properties`-HashMap ist es ebenso möglich, die Umwandlung in das für das Orca-Backend notwendige JSON-Format durch das OpenSource-Framework Genson (siehe [5]) durchführen zu lassen. Dort werden alle Properties automatisch anhand der Getter- und Setter-Methoden richtig gesetzt und eine Anpassung der JSON-Syntax ist nicht mehr notwendig. Es müsste ansonsten bei jeder Umwandlung in JSON explizit eine entsprechende Anpassung der Java- oder JSON-Objekte erfolgen, was einen deutlichen Mehraufwand bedeuten würde.

Bildlich gesprochen kann man sich die Entity-Klasse als ein Zahnrad vorstellen, das durch die Generierung des spezifischen Codes angetrieben wird, beide “Welten“ dadurch miteinander verbündet und “in Bewegung versetzt“. Die Entity-Klasse ist der zentrale *Verknüpfungspunkt zwischen generiertem Code und Plattformcode*.

4.3. Aufbau und Struktur einer generierten Entitätsklasse

Alle im Domänenmodell festgelegten Entitäten werden zu Java-Klassen generiert. Hierbei gibt es einige Besonderheiten, die zum besseren Verständnis bereits an dieser Stelle beschrieben werden.

Konstruktor

Im Konstruktor (siehe Listing 4.4, Zeile 3) jeder generierten Java-Klasse wird zunächst der EntityType durch entsprechenden Methodenaufruf gesetzt. Der EntityType wurde über das Domänenmodell festgelegt. Dadurch wird garantiert, dass bei jeder Instanzierung der EntityType korrekt gesetzt wird.

In den Zeilen 4-9 erfolgt ein ganz wichtiger Vorgang für die SQLitedatenbank-Schnittstelle. Die Entity-Klasse stellt zwei `HashMap<String, String>` und ein `HashSet<String>` bereit. Diese sind dafür gedacht, alle Attributnamen und deren Datentypen dort als Key-Value-Paare abzulegen. Anhand dieses Mechanismus ist eine generische Implementierung der Datenbankschnittstelle möglich. Alles Weitere hierzu betreffend wird in Abschnitt 4.6 erläutert.

Listing 4.4: Konstruktor einer generierten Klasse Person (Quelle: Eigener Quellcode)

```

1 ...
2 public Person() {
3     super.setType(1001);
4     super.entityPropertyDatatypes.put("name", "string");
5     super.entityPropertyDatatypes.put("vorname", "string");
6     super.entityPropertyDatatypes.put("geburtstag", "date");
7     super.entityPropertyDatatypes.put("standort", "double[]");
8     super.linkPropertyDatatypes.put("schreibtMit
9         nachricht", "string");
10    super.imagePropertyNames.add("profilbild");
11 }
```

Unterscheidung der Schnittstellen durch Methodennamen

Um eine für den/die Nutzer/in sichtbare Trennung der Schichten zu ermöglichen, wurden die entsprechenden Java-Methoden mit einem Namenszusatz versehen. Zum Laden einer

4. Modellgetriebene Entwicklung des Android SDKs

Person-Entität von dem Orca-Backend müsste folgende statische Methode der Klasse Person aufgerufen werden: `Person.BACKENDloadPersonFromBackendById(...)`. An erster Stelle der Schnittstellenmethoden erfolgt demnach entweder BACKEND, DATABASE oder MIX als Namenszusatz (für Methoden der kombinierten Schnittstelle).

Hierbei besteht nun natürlich die Gefahr einer “wilden“ Mischung der Methoden. Je nach festgelegter Attribute und Links wächst die Zahl vorhandener Methoden rapide. Das Ganze wäre auch durch eine zusätzliche Parserregel über das Domänenmodell abfragbar gewesen. Letztendlich wurde das System anhand der Methodennamen beibehalten, da es auch durchaus denkbar ist, manche Entitäten in einem Projekt nur lokal zu speichern, andere hingegen nur im Backend. Um einen besseren Überblick zu behalten, wurden “selbstsprechende“ Methodennamen verwendet.

Link und Image

Werden im Domänenmodell `link-` oder `image-`Typen festgelegt, werden dafür automatisch entsprechende Methoden zum Speichern, Laden, Aktualisieren und Löschen angelegt. Hierbei ist zu beachten, dass diese nur über das Backend angelegt werden können. Für die SQLitedatenbank-Schnittstelle wird demnach keine dieser Methoden generiert.

Dokumentation

Ein wichtiges Ziel der Implementierung war es alle generierten Entitätsklassen vollständig per Java-Doc zu dokumentieren. Dadurch wird die Auswahl der benötigten Methode stark vereinfacht, was aufgrund der Vielzahl an zur Verfügung stehenden Methoden auch notwendig ist.

4.4. Plattform-Code

Der Plattformcode umfasst alle generischen Klassen für die Orca- und SQLitedatebank-Schnittstelle. Ebenso sind mehrere Hilfsklassen enthalten. Die Packagestruktur wurde wie folgt aufgeteilt:

- Der einheitliche Packagepfad für den gesamten Plattformcode ist `de.fhws.sdk.orca`. Darin werden die spezifischen Klassen auf weitere Packages aufgeteilt.

4. Modellgetriebene Entwicklung des Android SDKs

- helper: Hier befinden sich Hilfsklassen, u.a. für die Verwaltung von statischen Konstanten oder z.B. Klassen mit statischen Methoden, die die Umwandlung und Syntaxprüfung von JSON zu Java und umgekehrt durchführen.
- model: In diesem Package befinden sich alle Java-Abbilder der Orca-Backend-Datenstrukturen, wie z.B. die im Abschnitt zuvor beschriebene Entity-Klasse.
- network: Alle für den Zugriff auf das Orca-Backend benötigten AsyncTask-Klassen befinden sich hierin. Dabei sind alle nochmals nach den ausgeführten HTTP-Standardmethoden GET, POST, PUT, DELETE und den Kategorien entity, link und image unterteilt, je nachdem welcher Request zu welchem Zweck ausgeführt wird. Ebenso befinden sich hier alle generischen Callback-Interfaces, die für die AsyncTask-Klassen verwendet werden.
- database: Das Pendant zu network. Alle benötigten Klassen für die SQLitedatenbank-Schnittstelle sind dort abgelegt.

In den folgenden Abschnitten werden Bestandteile des Plattformcodes anhand der jeweiligen Schnittstelle erläutert.

4.5. Orca-Backend-Schnittstelle

4.5.1. Backendzugriffe mit AsyncTask

Bei der Implementierung der Orca-Schnittstelle wurde großer Wert darauf gelegt, dass alle Endpunkte - und damit auch alle Funktionen - angesteuert und genutzt werden können. Für jeden Endpunkt wurde eine jeweilige AsyncTask-Klasse angelegt. Insgesamt sind 37 AsyncTask-Klassen implementiert worden. Dadurch lässt sich der gesamte Funktionsumfang des Orca-Backends nutzen.

In jeder AsyncTask-Klasse wird prinzipiell immer das gleiche Schema abgearbeitet (in der `doInBackground`-Methode):

1. Die Ziel-URI wird erstellt (oder wurde übergeben).
2. Bei POST- und PUT-Requests erfolgt die Umwandlung in JSON.
3. Der HTTP-Request wird mit allen benötigten *Headern* und Angaben im *Body* durchgeführt.

4. Modellgetriebene Entwicklung des Android SDKs

4. Der *StatusCode* der *Response* wird geprüft. Es wird ermittelt, ob die Anfrage positiv durchgeführt werden konnte.
5. Bei GET-Requests erfolgt hier die Umwandlung von JSON in ein Java-Objekt.
6. Der *StatusCode*, eine *Error Message* sowie evtl. weitere Rückgabewerte werden dem *Callback* übergeben.

An dieser Stelle ist ein Verweis zu der bereits in einem vorherigen Abschnitt erläuterten Packagestruktur (siehe Package-Struktur) des generierten Codes notwendig. In das selbe Package werden auch spezifische `AsyncTask`-Klassen generiert. Da es sich um Spezialfälle (wie z.B. das Abfragen eines bestimmten Attributwertes eines Attributes vom Backend) handelt, konnte bei diesen wenigen Fallgestaltungen keine generische Implementierung dieser Klassen erfolgen. Dies soll hier nur der Vollständigkeit halber erwähnt werden. Auf die zugehörigen Spezialfälle wird aufgrund deren Komplexität nicht weiter eingegangen.

In jedem individuell generierten Package wird auch ein Unterpackage `persistence` angelegt. Darin befinden sich generierte Wrapperklassen für den Backend- und Datenbankzugriff. Für jede Entität wird je eine individuelle Wrapperklasse generiert. Diese Wrapperklassen sind sozusagen eine Zwischenschicht zwischen den generierten Entitätsklassen und den `AsyncTask`-Klassen. `AsyncTask`-Klassen müssen meistens sehr viele Parameter zur Laufzeit übergeben werden, damit eine Abarbeitung möglich ist. Um diese Parameter zu kapseln, werden diese Wrapperklassen verwendet. Prinzipiell könnten diese auch außen vor gelassen werden. Diese Verfahrensweise hat sich bei der Implementierung des Codegenerators allerdings als sehr vorteilhaft erwiesen, da diese Zwischenschicht viele konstanten Parameter wie *Appname* und *Apikey* „abfängt“. In Abbildung 4.2 ist vereinfacht der Ablauf beim Aufruf einer Schnittstellenmethode aufgeführt.

4.5.2. Handhabung von Fehlern

Alle auftretenden Exceptions werden hierbei abgefangen. Der Gedanke dieses Konzeptes ist, dass es kaum Sinn machen würde, wenn der/die Entwickler/in manuell Exceptions behandeln müsste. Während der `doInBackground`-Methode können z.B. Fehler bei der JSON-Umwandlung auftreten.

Wie sollte auf eine entsprechende Exception reagiert werden? Der Grundgedanke aller Schnittstellen (des entwickelten Android SDKs) ist, dass alle Umwandlungen verdeckt im Hintergrund durchgeführt werden. Der/die Entwickler/in soll nur auf Java-Objekten programmieren, von einer Umwandlung in das JSON-Format soll er/sie nichts mitbekommen.

4. Modellgetriebene Entwicklung des Android SDKs

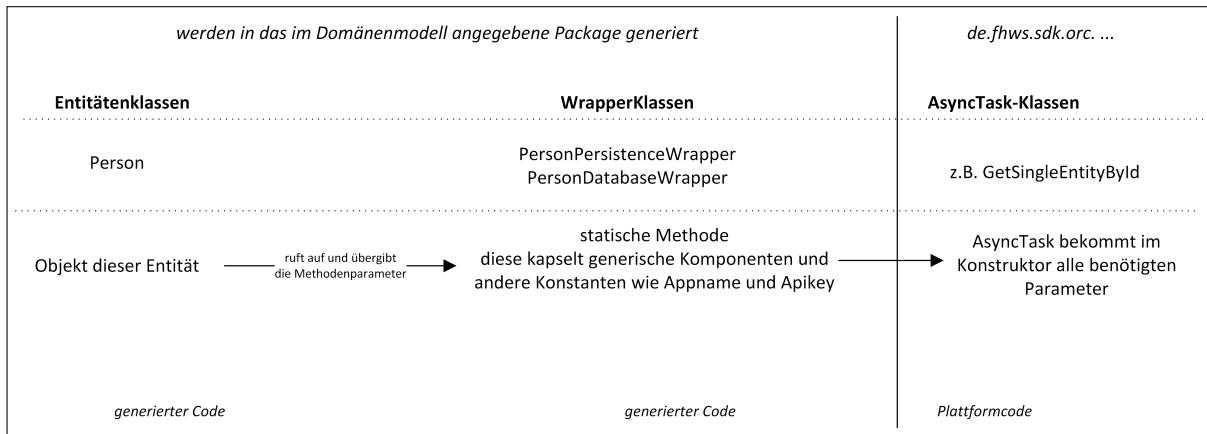


Abbildung 4.2.: Wrapperklassen vereinfachen die Verwendung generischer Klassen
(Quelle: Eigene Darstellung)

Auch können bei manchen AsyncTask-Klassen bis zu acht verschiedene Exceptions auftreten. Der Komfort, nur eine Methode für einen Backendzugriff aufrufen zu müssen, würde schwinden, wenn so viele Exceptions manuell abgefangen werden müssten. Deshalb werden alle auftretenden Exceptions innerhalb der AsyncTask-Klassen abgefangen, der StatusCode wird -1 gesetzt und anhand der errorMessage erfolgt eine Mitteilung, welcher Fehler aufgetreten ist. Anhand der einfachen Abfrage im Callback, ob der StatusCode -1 ist, können dadurch Fehler festgestellt werden.

4.5.3. Verwendung von Callbacks

Die Verwendung eines Callbacks bei jeder AsyncTask-Klasse ist für alle Methoden, die auf der Orca-Schnittstelle beruhen, von großer Bedeutung. Wie soeben erwähnt, können darüber StatusCode und eine errorMessage abgerufen werden. Je nach Request ist es auch möglich über das Callback die abgefragte Entität als Java-Objekt zu erhalten und auf dem Main Thread zu nutzen. Es sind verschiedene Callback-Interfaces implementiert worden, um je nach Request auch die entsprechenden Objekte abfragen zu können. Die meisten Callbacks sind auch generisch und erwarten als generischen Parameter ebenso eine Klasse vom Typ Entity.

Zum besseren Verständnis soll ein Laden einer Person-Entität vom Backend mit der ID 700 gezeigt werden. Der entsprechende Methodenaufruf mit Callback ist in Listing 4.5 zu sehen.

4. Modellgetriebene Entwicklung des Android SDKs

Listing 4.5: Methode der Orca-Backend-Schnittstelle mit Callback (Quelle: Eigener Quellcode)

```
1 ...
2 Person.BACKENDloadPersonFromBackendById(700, new
3           IReturnValueCallback<Person>() {
4
5             @Override
6             public void onComplete(Person entity, int statusCode, String
7             errorMessage) {
8
9               // entity ist vom Backend abgefragte Entity als Java-Objekt
10              // hier kann entity-Objekt auf dem Main Thread verwendet werden
11            }
12          });
13 ...
```

Alle Rückgabewerte können in der `onComplete`-Methode auf dem Main Thread verwendet werden. Allerdings kann es sein, dass `entity` den Wert `null` hat, wenn z.B. zu einer Suchanfrage kein Ergebnis gefunden wurde. Es ist deshalb ratsam, eine vorherige Prüfung durchzuführen, bevor das Objekt direkt verwendet wird.

4.6. SQLitedatenbank-Schnittstelle

Das Ziel bei der SQLitedatenbank-Schnittstelle war auch möglichst viel Code generisch zu implementieren. Da dies bei der Orca-Backend-Schnittstelle so gut funktionierte, wurde diese dort verwendete “Schichtenbildung“ mit Wrapperklassen analog verwendet. Ebenso wichtig war auch, dass die über das Domänenmodell festgelegten Entitäten ohne manuelle Anpassung zur SQLitedatenbank kompatibel sind. Das Domänenmodell soll keine Modifikationen benötigen, damit die SQLitedatenbank-Schnittstelle verwendet werden kann.

4.6.1. Datenbankzugriffe

Der Grundgedanke ist, dass für alle Entitäten eine einheitliche Datenbank erstellt wird. Diese wird mit dem *Appnamen* eindeutig benannt. Innerhalb dieser Datenbank wird für jeden Entitätentyp eine eigene Tabelle mit allen *möglichen* Attributen erstellt.

Hier traten schon die ersten Schwierigkeiten auf. Beim erstmaligen Speichern einer Entität muss zunächst geprüft werden, ob eine entsprechende Tabelle vom zugehörigen

4. Modellgetriebene Entwicklung des Android SDKs

Entitätstyp überhaupt in der Datenbank schon vorhanden ist. Ist keine Tabelle vorhanden, würde normalerweise eine Exception geworfen werden. Demnach hätte der/die Entwickler/in manuell die Tabelle erstellen müssen. Das wäre allerdings keine komfortable Handhabung. Insbesondere im Vergleich zur Backendschnittstelle müsste hier dann viel mehr beachtet werden.

Deshalb ist eine andere Herangehensweise gewählt worden. Für alle Datenbankzugriffe ist eine generische Klasse `DatabaseHandler<T extends Entity>` im Package `de.fhws.sdk.orca.database` vorhanden. Alle Datenbankzugriffe wie Speichern, Lesen, Ändern und Löschen erfolgen über diese. Nur anhand des Klassentyps können dort die Zugriffe gesteuert werden. Hierbei sollen beispielhaft die Vorgänge beim Speichern und Laden einer Entität gezeigt werden. Das Aktualisieren und Löschen von Entitäten verwendet die selben Verfahren, die bei den anderen beiden Vorgängen angesprochen werden. Sie werden deshalb nicht näher erläutert.

Speichern

Wird eine Entität zum ersten Male abgespeichert, werden mehrere Tabellen angelegt. Da alle Tabellen in der gleichen Datenbank angelegt werden, ist der Entitätsname als Tabellename unzureichend. Das Domänenmodell erlaubt nämlich die Festlegung von gleichnamigen Entitäten in unterschiedlichen Packages. Es könnte somit zu doppelten Tabellen für unterschiedliche Entitätstypen kommen. Als Tabellename für die Tabelle gleicher Entitäten wurde deshalb eine Kombination aus Packagennamen und Entitätennamen gewählt: z.B. `de_test_orcasdk_person`. Der Vorteil hiervon ist, dass ein generischer Abruf von Packagennamen und Klassennamen sehr einfach möglich ist. Entsprechende Methoden werden von Seiten Java standardmäßig zur Verfügung gestellt (siehe [8], Seite 495).

Entitäten können auch mehrere Links besitzen. Zur einfacheren Handhabung der Tabellen wurde entschieden, dass alle Links einer Entität in einer gesonderten Tabelle gespeichert werden. Der Tabellename für diese kann auch mit den von der Java-Klasse `Class` angebotenen Methoden ganz einfach erzeugt werden, denn es wird an den o. g. Tabellennamen der Entitätentabelle einfach nur „`_link`“ angefügt.

Es werden deshalb mindestens zwei Tabellen für jeden Entitätentyp benötigt: Eine Tabelle für die Entitäten selbst, eine weitere für die Links der Entitäten. Beide Tabellen sind dabei über die Entity-ID verknüpft (Primärschlüssel bei Entitätentabelle, Fremdschlüssel bei Linktabelle).

Erstellung des Tabellenschemas

Allerdings müssen beim Anlegen einer Tabelle auch die zugehörigen Spalten, also die Attribute mit deren Datentypen angelegt werden. Hier stand der generische Ansatz vor einer großen Hürde. Beim Speichern einer Entität wäre es möglich gewesen, die ganze Entität auszulesen und vor dem Speicherbefehl alle Attribute mit deren Datentypen per Java-Reflection zu ermitteln. Allerdings wäre dieser Ansatz sehr aufwendig und fehleranfällig. Der generische Ansatz würde dadurch wesentlich mehr Code benötigen. Auch soll der Einsatz von Reflection umgangen werden, wenn es andere (einfachere) Mittel und Wege gibt (siehe [8], Seiten 886-887).

Denkbar wäre gewesen, für jede Entitätsklassen eine individuelle Klasse zu generieren, die die benötigten Informationen aus dem Domänenmodell abfrägt. Hierbei wurde festgestellt, dass dieser Ansatz auch zu einer großen Menge an generiertem zusätzlichen Code geführt hätte. Deshalb wurde eine Mischlösung konzipiert.

Die benötigten Informationen (Attributnamen und deren Datentyp) werden über das Domänenmodell ausgelesen und anhand des Konstruktors in eine `HashMap<String, String>` übertragen (siehe hierzu Abschnitt 4.3). `entityPropertyDatatypes` verwaltet diese Informationen für die Entity-Properties, analog dazu verwaltet `linkPropertyDatatypes` diese für alle Link-Properties. Das hat mehrere Vorteile. Es wird wirklich nur der Code generiert, der tatsächlich benötigt wird. Zusätzlich ist es dadurch möglich, dass eine einfache Abfrage dieser Werte ermöglicht wird.

Beim Speichern einer Entität muss die zu speichernde Entität an die entsprechende Datenbankmethode übergeben werden. Innerhalb dieser Entität sind die o. g. HashMaps bereits entsprechend gesetzt. Diese werden ausgelesen und damit die Tabellenschemen erstellt.

Datentypen

Beim Abspeichern müssen auch manche spezielle Anpassungen bestimmter Datentypen erfolgen. Angenommen eine Entität hat ein Attribut vom Typ `Double[]` zum Abspeichern von Koordinationsdaten, dann müssen beide Werte aus dem Array gelesen und in je einer eigenen Spalte abgelegt werden. Auch ein `boolean`-Wert kann nicht einfach übernommen werden. Dieser wird in einen `Integer`-Wert umgewandelt (0 oder 1).

Laden

Beim Laden von Entitäten aus der Datenbank ergaben sich zusätzliche Fallstricke, die den generischen Ansatz noch erschweren. Als Ergebnis einer entsprechenden Abfrage wird immer ein Cursor-Objekt zurückgegeben. Darin sind alle gefundenen Entitäten als Zeilen (Datensätze) enthalten. Jede Zeile muss ausgelesen und in ein Java-Objekt umgewandelt werden.

Zusätzliche Erschwernis des generischen Ansatzes

Im Vergleich zum Speichern wird hier der jeweiligen Methode KEIN Entitäten-Objekt mitgegeben. Aufgrund der generischen Gestaltung wird hier nur ein Parameter einer Java-Klasse vom Typ `Class` übergeben. Die beim Speichern angesprochenen HashMaps können hier nicht verwendet werden. Dies ist problematisch, denn die darin enthaltenen Informationen zu dem jeweiligen Datentyp eines Attributes werden auch hier explizit benötigt. Zum Abrufen des entsprechenden Attributwertes muss nämlich eine vom zugehörigen Datentyp abhängige Methode aufgerufen werden (z.B. `getInt(...)`).

Beim Anlegen der Tabellen musste deshalb ein zusätzlicher Aspekt beachtet werden. Die in jedem Entitäten-Objekt enthaltenen HashMaps zur Feststellung von Attributnamen und Datentypen müssen nicht nur über ein Objekt eines bestimmten Entitätstyps abrufbar sein. Es muss auch möglich sein anhand des Java-Klassentyps diese Informationen zu ermitteln.

Aus diesem Grund werden beim erstmaligen Speichern einer Entität noch zwei weitere Tabellen angelegt, in denen die unter Abschnitt 4.6.1 genannten HashMaps gespeichert werden. Als Tabellenname wird dabei der Name der Entitäten-Tabelle mit dem Zusatz `_datatype` bzw. `_link_datatype` verwendet. So sind auch diese Tabellennamen immer eindeutig und mit Hilfe der Klasse `Class` abrufbar. Anhand dieser Klasse kann der Packagename und der Name der Entität gewonnen werden. Daraus sind die Namen der Tabellen rekonstruierbar, wodurch ein Laden der Tabellen ermöglicht wird. Die o. g. Tabellen enthalten die Attributnamen und Datentypen als Key-Value-Paare. Es sind deshalb nur Strings in diesen beiden Tabellen gespeichert. Auch werden beide Tabellen nach erfolgreicher Speicherung nicht mehr geändert oder Datensätze hinzugefügt.

Insgesamt werden somit vier Tabellen für jeden Entitätstyp beim erstmaligen Speichern angelegt. Abbildung 4.3 zeigt eine vereinfachte Übersicht dieser.

Alle in Abschnitt 3.3 spezifizierten Anforderungen werden wie gefordert mit dieser Konzeption ermöglicht.

4. Modellgetriebene Entwicklung des Android SDKs

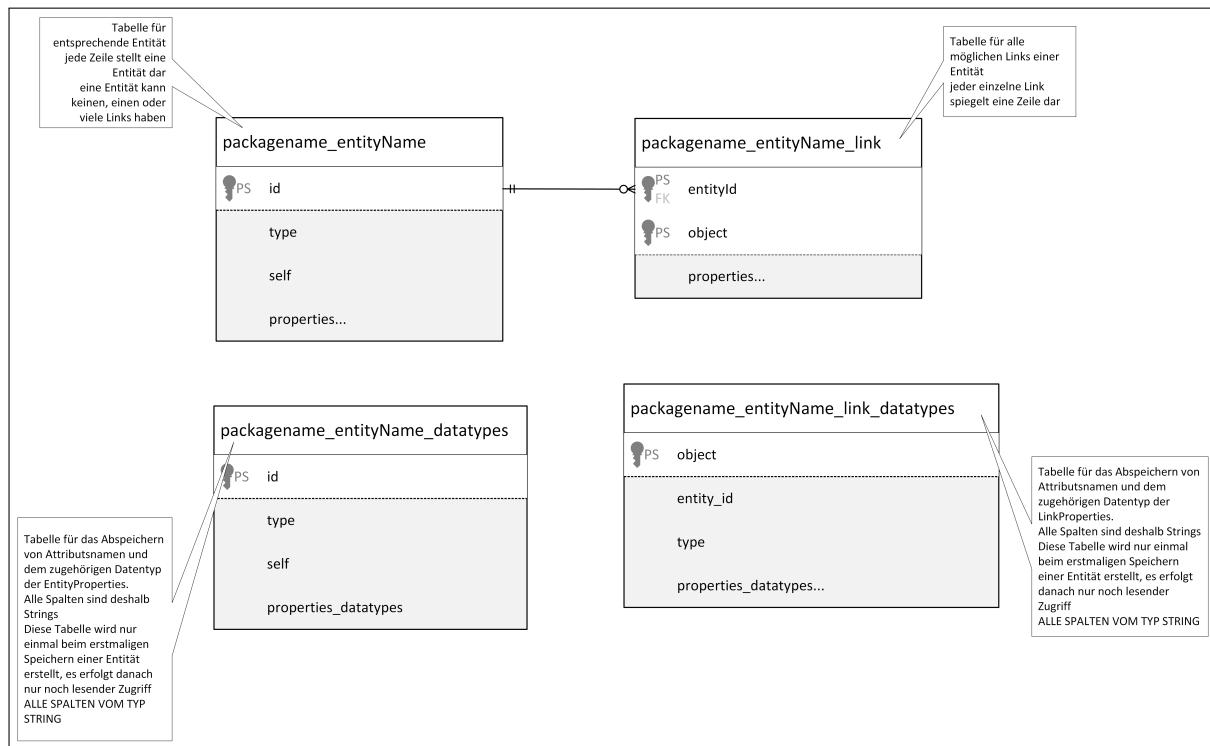


Abbildung 4.3.: Beim erstmaligen Speichern angelegte Tabellen (Quelle: Eigene Darstellung)

4. Modellgetriebene Entwicklung des Android SDKs

Der allgemeine Ablauf beim Laden ist demnach folgender: Zuerst werden die Tabellen mit den Datentypinformationen geladen. Dann erfolgt die eigentliche Suchabfrage, wo ein Cursor-Objekt erhalten wird. Dieses ist mit den Datentypinformationen entsprechend vollständig zu durchlaufen und die Umwandlung in die korrekten Datentypen kann erfolgen. Diese Prozedur erfolgt für die Entitäten-Tabelle und deren Link-Tabelle. Letztendlich ist noch ein entsprechendes Java-Objekt zu erstellen und zurückzugeben.

4.6.2. Fehlerbehandlung

Zugriffe auf die Datenbank müssen nicht wie bei der Orca-Schnittstelle auf einen separaten Thread ausgelagert werden. Diese können auch direkt auf dem Main Thread erfolgen. Eine Fehlerhandhabung wie bei der Orca-Schnittstelle (per Callback) macht in diesem Falle kaum Sinn und würde eher für Verwirrung sorgen.

Alle Methoden dieser Schnittstelle werfen deshalb Exceptions, falls Fehler beim Zugriff auf die Datenbank oder anderer Art auftreten. Dabei wird zumeist die Fehlermeldung der Exception individuell angepasst. Möchte man z.B. eine Entität aus der Datenbank laden, die dort nicht gespeichert ist, würde eine Exception mit entsprechend angepasster Fehlermeldung geworfen werden.

4.7. Sonderfall: kombinierte Schnittstelle

Diese Schnittstelle wurde bewusst zuletzt implementiert. Da von Anfang an ein wichtiges Konzept die Kompatibilität der Orca- und SQLitedatenbank-Schnittstelle war, wurde durch die erfolgreiche Umsetzung dieser der Grundstein für diese kombinierte Schnittstelle gelegt. Das Ziel war weiterhin den zu generierenden Code gering zu halten. Es sollte bestenfalls auch kein weiterer Plattformcode zusätzlich hinzukommen.

4.7.1. Realisierung

Der Fokus war deswegen gleich auf eine “tatsächliche“ Kombination der vorherigen Schnittstellen gelegt. Innerhalb der generierten Klassen werden die beiden anderen Schnittstellen anhand deren Methoden so kombiniert, dass eine entsprechende Funktionalität erreicht wird.

Genau nach diesem Prinzip wurde die Umsetzung dieser Schnittstelle realisiert. Wie in der Spezifikation gefordert, erfolgt immer zuerst ein Zugriff auf das Orca-Backend, ge-

4. Modellgetriebene Entwicklung des Android SDKs

folgt von einem Zugriff zur SQLitedatenbank-Schnittstelle. Die Vor- und Nachteile dieser Konstruktion waren letztlich auch verantwortlich, diese Schnittstelle so umzusetzen.

+ Codeumfang wird kaum vergrößert Keine einzige zusätzliche Klasse muss implementiert werden. Es sind lediglich Methoden direkt in die Entitätsklassen zu generieren, die den Methodenaufruf der Orca-Backend- und SQLitedatenbank-Schnittstelle kapseln und steuern.

+ Komplexität steigt in überschaubarem Maße Durch die Wiederverwendung des Codes der anderen Schnittstellen steigt die Komplexität kaum an.

+ Zeitersparnis Eine Implementierung einer kompletten neuen Schicht hätte enorm Zeit gekostet.

+ Änderungen müssen nur an einer Stelle vorgenommen werden Wird eine der beiden vorherigen Schnittstellen geändert (z.B. Fehler werden beseitigt), wirkt sich dies auf die kombinierte Schnittstelle automatisch mit aus.

+ Daten bleiben konsistent Es erfolgt immer zuerst ein Zugriff zum Backend. Dort wird die eindeutige ID für jede Entität vergeben. Ebenso erfolgt die Linkerstellung nur über das Backend. Lokal befindet sich dadurch auch immer der aktuellste Stand. Dieser Punkt wäre über eine neue Schnittstelle auch realisierbar gewesen, allerdings nicht in Bezug auf die oben bereits genannten Punkte.

+ Fehleranfälligkeit sinkt Beide vorherigen Schnittstellen sind zueinander kompatibel. Durch eine Neuimplementierung wäre die Gefahr groß, dass es zu Fehlern kommt. Durch die unveränderte Verwendung der beiden vorherigen Schnittstellen sinkt die Fehleranfälligkeit.

+ voller Funktionsumfang der beiden anderen Schnittstellen Beide vorherigen Schnittstellen erfüllen die jeweiligen an sie gestellten Anforderungen. Entsprechend leistungsfähig sind diese auch. Ebenso wird dadurch gewährleistet, dass beide kompatibel zueinander sind. Es muss demnach keine aufwändige Umwandlung der Entitäten erfolgen. Beide “arbeiten“ mit Java-Objekten.

4. Modellgetriebene Entwicklung des Android SDKs

- **kein Rollback** Die kombinierte Schnittstelle hat prinzipiell nur einen Nachteil - dafür allerdings einen schwerwiegenden. Problematisch ist diese Konzeption, wenn nach dem erfolgreichen Backendzugriff (Daten wurden dort somit geändert) die Aktualisierung der SQLitedatenbank fehlschlägt. Es ist kein Rollback zum vorherigen Zustand möglich.

Die Umsetzung eines entsprechenden Mechanismus hätte bedeutet, entweder eine eigenständige Implementierung dieser Schnittstelle vorzunehmen oder eine aufwändige Zwischenlösung zu bewerkstelligen. Dies hätte die meisten o. g. positiven Aspekte verworfen. Auch wurde berücksichtigt, dass es an diesem Punkt kaum zu Fehlern kommen kann. Fehler würden nur auftreten, wenn auf die SQLitedatenbank nicht zugegriffen werden kann oder wenn es zu Umwandlungsfehler der Java-Entitäten kommen würde. Da beide Schnittstellen zueinander kompatibel sind, wird davon ausgegangen, dass der Eintritt von Fehlern an dieser Stelle sehr unwahrscheinlich ist. Bei praktischen Tests konnte kein Sachverhalt konstruiert werden, der bei normalem Ablauf der kombinierten Schnittstelle zu einem Fehler an dieser Stelle geführt hat.

Aufgrund dessen, und weil die positiven Aspekte deutlich überwiegen, wurde diese Realisierung der kombinierten Schnittstelle beibehalten.

4.7.2. Workflow

Zum besseren Verständnis des genauen Ablaufs eines Methodenaufrufes dieser Schnittstelle wurde die Abbildung 3.3 überarbeitet und angepasst. In Abbildung 4.4 ist der tatsächliche Workflow abgebildet.

4.7.3. Fehlerbehandlung

Alle Methoden der kombinierten Schnittstelle verwenden zunächst eine Orca-Backend-Schnittstellenmethode. Demzufolge wird zuerst immer ein `AsyncTask` abgearbeitet. Die Methode für die Ansteuerung der SQLitedatenbank darf demnach erst in einer `Callback`-Methode gestartet werden.

Allein diese kurze Beschreibung des Workflows lässt schon darauf schließen, dass Methoden der kombinierten Schnittstelle zum Teil sehr komplex sind. U.U. sind sogar mehrere `Callback`-Methodenaufrufe ineinander verschachtelt.

Eine Fehlerhandhabung per Exceptions würde hier keinen Sinn machen. Daher wird auch hier das gleiche Verfahren wie bei der Orca-Schnittstelle verwendet. Der/die Entwickler/in muss beim Aufruf einer solchen Methode ein `Callback`-Objekt übergeben und kann darüber die gewohnten Parameter abfragen.

4. Modellgetriebene Entwicklung des Android SDKs

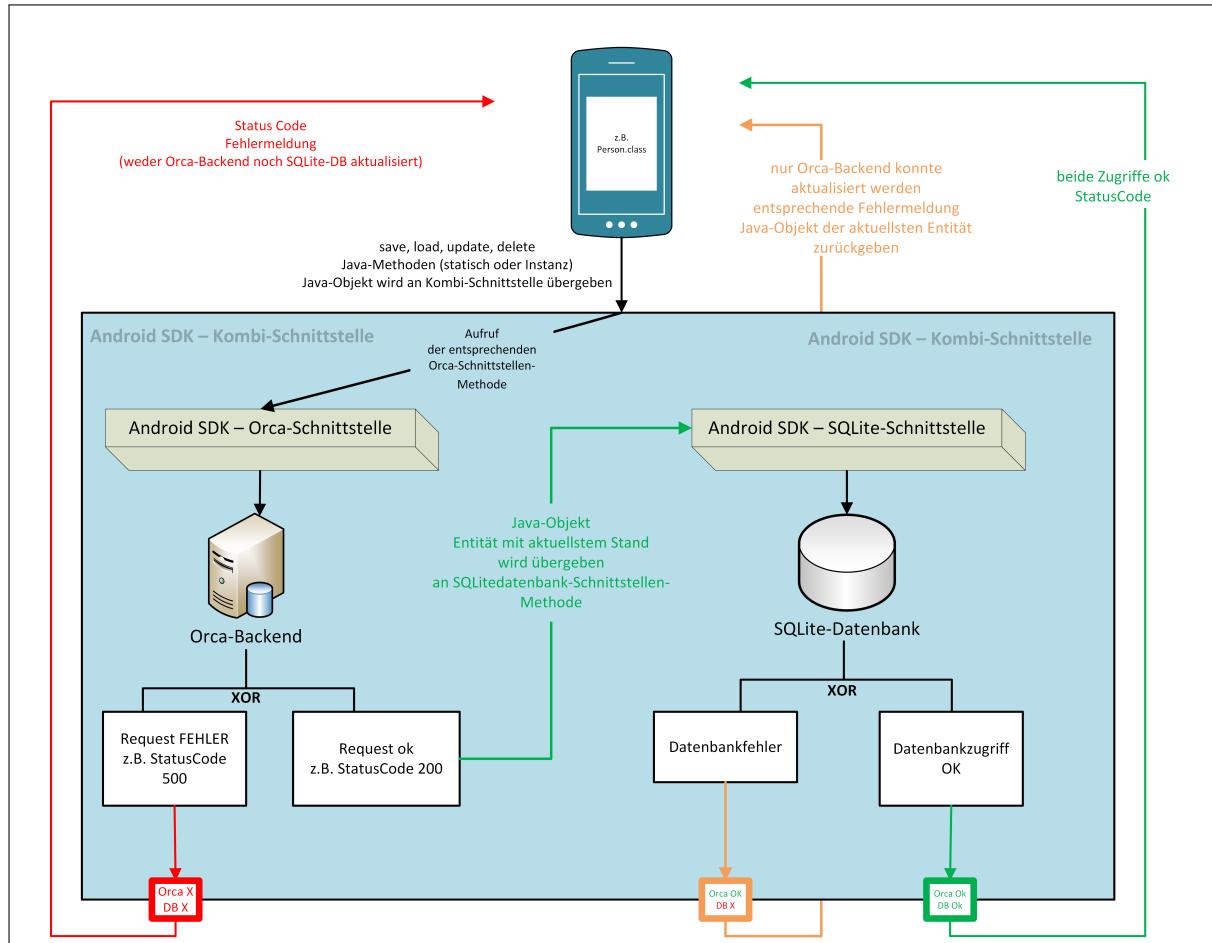


Abbildung 4.4.: Tatsächlicher Workflow der kombinierten Schnittstelle (Quelle: Eigene Darstellung)

4.8. Qualitätssicherung

Um die Funktionalität der Schnittstellen zu gewährleisten, wurden alle aufrufbaren Methoden mehreren Praxistests durchzogen. Insgesamt wurden während der gesamten Erstellung dieser Arbeit über 700 Entitäten für das Testen verwendet.

Es soll hier ausdrücklich erwähnt werden, dass aufgrund der Komplexität und der Vielzahl an verschiedenen Methoden keine wirklich umfassenden Tests durchgeführt wurden und keine Testkonzepte erstellt werden konnten. Insbesondere fehlte die Zeit für automatisiertes Testen. Auch konnten die genannten Praxistests nur auf einem Windows-System (8.1) und auf einem Android Smartphone (Android Version 4.3) durchgeführt werden.

4.9. Zusammenfassung

In diesem Kapitel wurden grundlegende Konzepte der Implementierung des Android SDKs vorgestellt.

Die mit Xtext verfasste DSL ermöglicht die Festlegung eines Domänenmodells, worin Entitäten definiert werden können, die per Xtend-Codegenerator automatisch erstellt werden.

Alle generierten Entitäten sind vollständig kompatibel zu den in Kapitel 3 herausgearbeiteten Spezifikationen der Orca-Backend- und SQLitedatenbank-Schnittstelle.

Die kombinierte Schnittstelle ist eine Kombination der beiden anderen Schnittstellen. Für die Umsetzung dieser musste keine zusätzliche (generische) Klasse implementiert werden. Alle Funktionen werden direkt in die generierten Entitätsklassen integriert. Es wird bei allen Methoden dieser Schnittstelle immer zuerst der Zugriff auf das Orca-Backend durchgeführt, anschließend auf die SQLitedatenbank. Nachteilig ist bei dieser Konzeption nur, dass es theoretisch zu inkonsistenten Datenständen kommen kann, falls nach erfolgreicher Backendaktualisierung der SQLitedatenbankzugriff fehlschlägt. Dieses Risiko wird aufgrund der überwiegenden Vorteile dieser Umsetzungsvariante in Kauf genommen.

Eine Qualitätssicherung wurde regelmäßig in Form von praktischen Tests durchgeführt. Alle Methoden wurden dabei mit gültigen wie auch fehlerhaften Daten getestet. Es war hierbei allerdings nur möglich, eine Hardwarekonfiguration sowie ein Smartphone heranzuziehen.

5. Evaluierung der Ergebnisse

Im vorherigen Kapitel wurde auf die Implementierung des generierten Codes und des Plattformcodes eingegangen. Dort wurde bereits erwähnt, dass die in Kapitel 3 festgelegten Spezifikationen eingehalten wurden. Dadurch ist allerdings nach wie vor unklar, ob durch einen Einsatz des SDKs auch wirklich Vorteile (insbesondere in zeitlicher Hinsicht) im Rahmen einer Appentwicklung erzielt werden können und sich die Verwendung lohnt.

In diesem Kapitel erfolgt deshalb eine Validierung und es wird insbesondere auf den gewünschten Effekt einer Zeitersparnis und einer Erleichterung der Implementierung eingegangen. Darüber hinaus wird ein zusätzlicher Fokus auf die Verwendung des modellgetriebenen Ansatzes bei Android-Entwicklungen gelegt.

5.1. Zeitersparnis

Der wichtigste Vorteil der Nutzung des entwickelten Android SDKs soll einen (erheblichen) Zeitgewinn bei der App-Entwicklung darstellen. Die Einrichtung der Entwicklungsumgebung sollte kaum Zeit kosten dürfen. Eine Einarbeitung bezüglich des Orca-Backend und der SQLitedatenbank soll fast vollständig wegfallen.

Zur besseren Einschätzung soll hier die geschätzte zu investierende Zeit, die vor der eigentlichen Appcodierung investiert werden muss, bei Verwendung mit und ohne das SDK gegenübergestellt werden. Es soll davon ausgegangen werden, dass die zu entwickelnde App eine Anbindung an ein Backend und an die lokale SQLitedatenbank benötigt.

In Tabelle 5.1 ist deutlich erkennbar, dass bei einer Nutzung des SDKs deutlich weniger Zeit „verloren“ geht und prinzipiell sofort mit der individuellen Appcodierung begonnen werden kann.

5. Evaluierung der Ergebnisse

Tabelle 5.1.: Vergleich des zeitlichen Aufwands VOR Schreiben des Appcodes (Quelle: Eigene Schätzungen¹⁾)

Themenbereich	ohne SDK	mit SDK
Einarbeitung Orca-Backend	ca. 40 Std.	entfällt
Einarbeitung SQLiteDB	ca. 15 Std.	entfällt
Implementierung von Schnittstellen	ca. 60 Std.	entfällt
Einrichtung des SDKs (s. Anhang C)	entfällt	ca. 2 Std.

Bedenkt man nun, dass das SDK appübergreifend verwendet werden kann, fallen die Zahlen noch viel mehr ins Gewicht, denn es müsste evtl. für jede App abermals eine entsprechende Vorarbeit geleistet werden. Das "bremst" letztendlich die gesamte Entwicklungsdauer erheblich.

Durch Verwendung des SDKs kann enorm viel Zeit gespart werden. Um sich selbst davon einen Eindruck machen zu können, wurde unter Anhang C eine Schritt-für-Schritt-Anleitung zur Einrichtung des SDKs in einer Entwicklungsumgebung erstellt. Großzügigen Schätzungen nach dauert die Einrichtung ca. zwei Stunden. Werden die dort genannten Schritte genau befolgt, kann das SDK umgehend genutzt werden - ohne jegliche Vorkenntnisse über das Orca-Backend und der SQLitedatenbank.

Das Ziel, eine Zeitersparnis vor der eigentlichen individuellen Appcodierung bei der Appentwicklung durch Verwendung des SDKs zu erreichen, wurde demnach erfüllt.

5.2. Bewertung des Codeumfangs

Ein wichtiger Grundgedanke bei der Erstellung des SDKs war es, einen generischen Ansatz zu konzipieren, womit der zu generierende Code - und somit der Gesamtumfang des ganzen Codes - minimal gehalten werden sollte.

Um das Ganze zahlenmäßig greifbar zu machen, soll hier ein Beispielvergleich durchgeführt werden. Als Gegenstück zu der gewählten generischen Implementierung (mit minimalem Umfang an generiertem Code) soll eine Variante betrachtet werden, bei der möglichst viel Code generiert wird, dafür aber der Platformcode gering gehalten wird. Diese zweite Variante wurde ganz zu Beginn der Implementierung auch in Erwägung gezogen.

¹⁾Die Zeitangaben beruhen auf der Einarbeitungsdauer, die im Rahmen dieser Arbeit für das jeweilige Themengebiet ca. aufgewendet wurde.

Codeumfang bei minimalem generierten Code

Nach diesem Grundsatz wurde das SDK erstellt. Zunächst soll die genaue Zusammensetzung des Plattformcodes und des generierten Codes untersucht werden.

Als Beispiel wird die in Listing 5.1 aufgeführte Entität Person mit dem Android SDK generiert.

Listing 5.1: Beispiel Domänenmodell (Quelle: Eigener Quellcode)

```

1 entity person{
2     TYPE = 100;
3     boolean verheiratet;
4     string name;
5     date geburtsdatum;
6     double bargeld;
7     double[] standort;
8     image profilbild;
9     link schreibtMit person{
10         boolean online;
11         string nachricht;
12         long linkid;
13         double bewertung;
14         double[] location;
15         date datum;
16     }
17 }
```

In den Tabellen 5.2 und 5.3 ist die Anzahl der jeweiligen Klassen und der ermittelten Lines of Code (LOC) ersichtlich, die nach der Generierung o. g. Entität gezählt werden konnten.

Tabelle 5.2.: Umfang des Plattformcodes bei minimaler Codegenerierung (Quelle: Eigene Berechnungen)

Plattformklassen	Anzahl	LOC
generische	45	ca. 8000
sonstige	14	ca. 1500

Tabelle 5.3.: Umfang des generierten Codes für Entität Person bei minimaler Codegenerierung (Quelle: Eigene Berechnungen)

generierte Klassen	Anzahl	LOC
individuell	6	ca. 3400
sonstige	15	ca. 1700

5. Evaluierung der Ergebnisse

Bei der generierten Klasse Person fallen auf Seiten des Plattformcodes ca. 9500 LOC und auf Seiten des generierten Codes ca. 5000 LOC an, insgesamt somit ca. 15000 LOC. Zu beachten ist hierbei, dass die Anzahl der in Tabelle 5.3 als “sonstige“ bezeichneten Klassen (hierbei handelt es sich um generierte Klassen, die von anderen Entitäten verwendet werden können) auch bei weiteren zu generierenden Entitäten nicht zunimmt. Wird nun eine zweite Entität mit gleichem Umfang an Attributen zusätzlich generiert, steigt der gesamte Codeumfang um ca. 3400 LOC.

Codeumfang bei minimalem Plattformcode

Beispielhaft wird nun angenommen, dass die in Tabelle 5.2 genannten generischen Klassen vollständig in den Generator übertragbar sind. In diesem Falle verringert sich der Plattformcode erheblich. Allerdings steigen die dadurch individuell zu generierenden Klassen deutlich an. Die Tabellen 5.4 und 5.5 zeigen hierfür entsprechende Schätzwerte.

Tabelle 5.4.: Umfang des Plattformcodes bei minimalem Plattformcode (Quelle: Eigene Berechnungen)

Plattformklassen	Anzahl	LOC
generische	0	0
sonstige	14	ca. 1500

Tabelle 5.5.: Umfang des generierten Codes für Entität Person bei minimalem Plattformcode (Quelle: Eigene Berechnungen)

generierte Klassen	Anzahl	LOC
individuell	51	ca. 11400
sonstige	15	ca. 1700

Hierbei ist erkennbar, dass der individuell zu generierende Code sehr stark zunimmt. Pro festgelegter Entität würden demnach ca. 11400 LOC generiert.

5. Evaluierung der Ergebnisse

Vergleich beider Varianten

In Tabelle 5.6 werden die zuvor ermittelten Werte gegenübergestellt.

Tabelle 5.6.: Gegenüberstellung des jeweiligen gesamten Codeumfangs in LOC (Quelle: Eigene Berechnungen¹)

Entitäten	min. gener. Code	min. Plattformcode
1	ca. 14600	ca. 14600
2	ca. 18000	ca. 26000
5	ca. 28200	ca. 60200
10	ca. 45200	ca. 117200

Bei der Variante mit minimiertem Plattformcode würde bei zunehmender Anzahl an Entitäten der Umfang des Codes förmlich explodieren, steigt hier der gesamte Codeumfang um ca. 11400 LOC je generierter Person-Entität (im Gegensatz zu ca. 3400 LOC bei der anderen Variante). Nur bei einer festgelegten Entität - was ein ziemlich unrealistischer Anwendungsfall wäre - halten sich beide Varianten die Waagschale.

Durch die generische Implementierung konnte erreicht werden, dass der generierte - und somit auch der gesamte - Code minimal gehalten wird, und sich somit Projekte bei Verwendung des SDKs nicht künstlich „aufblähen“. Zudem ist der Codeumfang nicht zu stark von der Anzahl der zu erstellenden Entitäten abhängig.

5.3. Handhabung des SDKs

In diesem Abschnitt soll die praktische Verwendung des SDKs anhand bestimmter Fallgestaltungen, die bei einer Appentwicklung auftreten, gezeigt und bewertet werden. Als Beispiel-Entität wird die in Listing 5.1 festgelegte Person herangezogen.

Auswahl einer Schnittstelle

Bei jeder generierten Entität sind alle drei Schnittstellen über Methodenaufrufe auswählbar. Welche Schnittstelle die entsprechende Methode anspricht, wird anhand des Methodennamens unterschieden. Alle Methoden, die das Orca-Backend ansprechen, begin-

¹ Angaben beziehen sich auf Tabellen 5.2 und 5.3 sowie Tabellen 5.4 und 5.5

5. Evaluierung der Ergebnisse

nen mit dem Wort “BACKEND“, die Methoden der SQLitedatenbank-Schnittstelle mit “DATABASE“ und “MIX“ kennzeichnet die Methoden der kombinierten Schnittstelle.

In der Eclipse-Entwicklungsumgebung hat dies den großen Vorteil, dass bei Eingabe eines Anfangsbuchstabens einer Methode und Verwendung des Shortcuts “*Shift + Leertaste*“ nur die Methoden angezeigt werden, die damit beginnen (siehe hierzu Abbildung 5.1, dort wird die Auswahl der kombinierten Schnittstelle gezeigt).

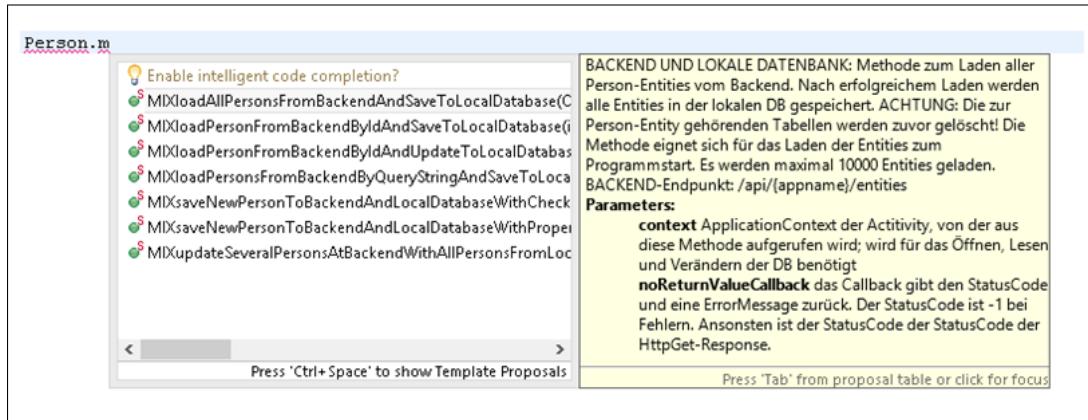


Abbildung 5.1.: Auswahl einer Methode in Eclipse (Quelle: Eigene Darstellung)

Die Auswahl einer der drei Schnittstellen ist somit sehr einfach möglich.

Allerdings können die teils sehr langen Methodennamen das “Finden“ der “richtigen“ Methode innerhalb einer Schnittstelle erschweren. Diese sind größtenteils zwar selbst-erklärend, doch einige Namen sind aufgrund ähnlicher Funktionen fast gleichlautend. Die für jede Methode vorhandene Java-Doc kann diesen negativen Aspekt größtenteils wieder ausmerzen.

Pro und Contra der Verwendung

Zur Fehlervermeidung bei einer Methodenauswahl werden bestimmte Methoden entweder nur als statische oder als Instanzmethode zur Verfügung gestellt. Dadurch werden Anwendungsfehler vorab schon vermieden.

Durch das Konzept, alle Funktionen per Methodenaufruf nutzen zu können, ist die Handhabung grundsätzlich sehr einfach und intuitiv. In Listing 5.2 wird eine Methode zum Speichern einer Person-Entität verwendet, die als Parameter die im Domänenmodell festgelegten Attribute erwartet. Fehlerhafte Datentypen oder falsche Attribute können schon gar nicht übergeben werden. Nach Methodendurchlauf ist die Entität im Backend und lokal gespeichert. Die ID und alle URIs, die vom Backend erstellt wurden, sind

5. Evaluierung der Ergebnisse

dabei auch in der lokalen Datenbank abgespeichert. Es soll hier verdeutlicht werden, dass dieser sehr komplexe Vorgang vollständig gekapselt wird, was für einen sehr hohen Komfort des SDKs spricht.

Listing 5.2: Eine Methode kapselt Backend und Datenbank vollständig (Quelle: Eigener Quellcode)

```
1 //komplette Kapselung von Backend und SQLiteDB
2 Person.MIXsaveNewPersonToBackendAndLocalDatabaseWithProperties(
    true, "Meier", new Date(), 290.45, new Double[]{33.4, 44.5}, this, new
    IReturnValueCallback<Person> () {
3
4     @Override
5     public void onComplete(Person entity, int statusCode, String
        errorMessage) {
6
7         //entity ist erstellt
8         //ist im Backend und in lokaler DB gespeichert
9         //die vom Backend vergebene ID kann abgefragt werden
10        //ist in SQLiteDB identisch
11        entity.getId();
12    }
13 } ;
```

Auch trägt der Callback-Mechanismus bei allen Methoden, die auf das Backend zugreifen, dazu bei, dass Laufzeitfehler vermieden werden. Grundsätzlich kann bei allen Methoden, die ein Callback-Objekt erwarten, auch null übergeben werden. Hierbei erfolgt trotzdem die Abarbeitung der Methode. Dies ist einerseits von Vorteil, denn dadurch kann der Callback-Mechanismus individuell verwendet werden.

Allerdings kann dies problematisch werden. Insbesondere bei der Verwendung von Instanzmethoden läuft man Gefahr einfach mit der gleichen Instanz “weiterzuarbeiten“. In Listing 5.3 ist ein Beispiel abgebildet.

Listing 5.3: Fehler bei nachfolgendem Methodenaufruf (Quelle: Eigener Quellcode)

```
1 //kein Compilerfehler! Beide Methoden werden aufgerufen.
2 person.BACKENDsaveThisPersonToBackendWithoutCheck(null);
3 person.BACKENDupdatePersonWithNewPropertiesAtBackendAndAtThisObject (
    null, "meier", null, null, null, null);
4 //FEHLER - Erste Methode ist noch nicht durchgelaufen,
5 //Speicherung im Backend ist beim Aufruf der zweiten Methode
6 //noch nicht beendet!
7 //Anwendung wird auch nicht beendet!
```

Besonders fatal an diesem Beispiel ist, dass die Anwendung weiterläuft und nicht abstürzt. Dieser Fehler (das Update ist u. U. nicht erfolgt) würde erst bei einer weiteren (späteren)

5. Evaluierung der Ergebnisse

Abfrage der Entität vom Backend mehr oder weniger zufällig auffallen. Der Vorteil, dass auf Instanzen gearbeitet werden kann, wird hier leider zu einem Nachteil. Besonders wenn es darum geht, mehrere aufeinanderfolgende Zugriffe zum Backend RICHTIG zu bewerkstelligen, kann der Code sehr umfangreich werden und es wird schwierig, die Übersicht zu behalten. Dies ist in Listing 5.4 gut erkennbar.

Listing 5.4: Korrekte Handhabung aufeinander aufbauender Methodenaufrufe (Quelle: Eigener Quellcode)

```
1 //so ist die korrekte Handhabung
2 person.BACKENDsaveThisPersonToBackendWithoutCheck(new
3           IReturnValueCallback<Person> () {
4
5             @Override
6             public void onComplete(Person entity, int statusCode, String
7             errorMessage) {
8
9               person.BACKENDupdatePersonWithNewPropertiesAtBackend
10              AndAtThisObject(null, "meier", null, null, null, null);
11
12           }
13       );
14 //entsprechende Aufrufe sind u.U. tief verschachtelt
```

Hier ist schon erkennbar, dass es zu tiefen Verschachtelungen kommen kann, wenn Methoden, die ein Callback-Objekt als Parameter erwarten, aufeinanderfolgend auszuführen sind. Es bedarf deshalb einer wohlbedachten Verwendung jener Methoden. Obwohl es sich um Instanzmethoden handelt, dürfen diese keinesfalls als “echte“ Instanzmethoden betrachtet werden.

Das Ziel einer einfachen Handhabung wird durch das Methodenkonzept auf intuitive Weise erfüllt. Alle Methoden kapseln vollständig die eigentlich angesteuerten Bezugspunkte. Bestimmte Methoden werden auch speziell nach den Angaben im Domänenmodell erstellt. Es werden z.B. die dort festgelegten Attribute als Methodenparameter erwartet. Dadurch ist ausgeschlossen, dass Attribute falsche Werte oder Datentypen erhalten.

Es bedarf jedoch großer Aufmerksamkeit des/der Entwicklers/Entwicklerin bei der Verwendung von Methoden des SDKs mit dem Callback-Mechanismus. Die einfache Handhabung verleitet u. U. zu einem leichtsinnigen Umgang, was schwerwiegende und auch schwer aufzudeckende Fehler verursachen kann.

5.4. Modellgetriebene Entwicklung unter Android

Für die modellgetriebene Entwicklung dieser SDK waren zeitlich aufwändige Vorarbeiten notwendig. Erst nachdem grundlegende Punkte des Orca-Backends herausgearbeitet waren, konnte mit einer Umsetzung des modellgetriebenen Paradigmas begonnen werden.

Der Einstieg in die Materie wurde erschwert, da es kaum Fachbücher gibt bzw. kaum Informationsmaterial im Internet gefunden werden konnte. Dementsprechend aufwändig war auch hier die Einarbeitung.

An diesem Punkt muss klargestellt werden, dass sich dieser Aufwand lohnen muss. Eine Verwendung macht demnach nur Sinn, wenn auf lange Sicht gedacht wird. Eine modellgetriebene Entwicklung von Grund auf für nur eine App zu konzipieren, ist den enormen zeitlichen Aufwand wohl kaum wert. Eine modellgetriebene Entwicklung muss klar auf eine Wiederverwendbarkeit der dadurch zu generierenden Daten ausgelegt werden.

Nachdem entsprechende Grundsteine gelegt waren, konnte relativ schnell eine DSL entwickelt werden, die die Spezifikationen des Orca-Backends abdeckt. Nachträgliche Korrekturen waren kein Problem. Auch die Implementierung des Generators war nach der endgültigen Festlegung der DSL kein großer konzeptioneller Aufwand, sondern vielmehr Schreibarbeit.

Die verwendeten Tools Xtext und Xtend können hier empfohlen werden. Es gab kaum Probleme bei der Einrichtung und der Verwendung. Es muss hier nochmals ausdrücklich erwähnt werden, dass von vornherein ein Fokus auf minimaler Codegenerierung lag. Dementsprechend schlank fällt auch die implementierte DSL (siehe Listing A.1) aus. Dies war von Vorteil, da dadurch eine tiefgehende Einarbeitung in Xtext und Xtend nicht notwendig war. Ohne Probleme konnten die Tools in eine Android-Entwicklungs-Umgebung integriert werden, wodurch sich weitere modellgetriebene Entwicklungen in diesem Kontext anbieten.

5.5. Validierung der Qualitätssicherung

Ein ganz wichtiger Punkt, der hier nicht unbeachtet bleiben darf, ist die noch unzureichende Qualitätssicherung des SDKs. Wie in Abschnitt 4.8 erwähnt, konnten zwar alle Methoden mehrmals getestet werden, jedoch ist der Umfang und die Qualität der Testverfahren als viel zu gering einzustufen.

Aufgrund der Masse an zur Verfügung stehenden Methoden, ist ein methodisches (evtl.

5. Evaluierung der Ergebnisse

auch automatisiertes) Testen notwendig. Hierfür sprechen mehrere Gründe:

- Alle Tests fanden nur in einer Ein-Nutzer-Umgebung statt.
- Das Verhalten bei Verwendung des SDKs in einer aufwändigen App (mit vielen Methodenaufrufen der Schnittstelle(n)) ist unklar.
- Tests konnten bislang nur auf einer Hardwarekonfiguration und nur auf einem Smartphone erfolgen.

Zum jetzigen Entwicklungsstand muss deshalb von einer unzureichenden Qualitätssicherung gesprochen werden.

5.6. Fazit

Das entwickelte SDKs ist nach dem Erkenntnisstand der im Rahmen dieser Arbeit durchgeführten Tests ein sehr leistungsfähiges Tool. Die Verwendung hängt nur davon ab, ob das eigentliche Datenmodell einer bestimmten App mit dem Domänenmodell des SDKs abzubilden ist. Prinzipiell sollte dies aber kein Problem darstellen, da alle gängigen Datentypen unterstützt werden.

Das SDK kapselt vollständig die Kommunikation zum Orca-Backend sowie zur SQLite-datenbank. Eine Einarbeitung fällt fast vollständig weg. Bei Nutzung des SDKs kann demnach gleich mit dem Programmieren des Appcodes begonnen werden. Dadurch entsteht ein immenser Zeitgewinn, der vollends in die eigentliche App investiert werden kann.

Auch trägt das auf Minimierung des generierten Codes ausgelegte generische Konzept Früchte. Dadurch kann der gesamte Codeumfang möglichst gering gehalten werden. Gerade im Kontext mit mobilen Applikationen ist es wünschenswert, dass diese an und für sich “kleineren“ Softwareprojekte überschaubar bleiben.

Eine Schnittstellenauswahl wird über klar bezeichnete Methodennamen vereinfacht. Dadurch werden die Methodennamen teils sehr lang und sind nicht immer auf den ersten Blick selbsterklärend. Die vorhandene Java-Doc zu jeder Methode kompensiert diesen Nachteil. Die Verwendung und Handhabung des SDKs ist praktisch und intuitiv, denn es gewährleistet durch einfache statische Methoden und Instanzmethoden Wahlfreiheit über alle drei Schnittstellen. Dadurch entsteht allerdings die Gefahr einer leichtsinnigen und zu häufigen Verwendung. Zur Vermeidung von Fehlern ist bei allen Methoden mit Callback-Mechanismus dieser zu verwenden. Dadurch kann es zu teils unübersichtlichen tiefen Code-Verschachtelungen kommen. Kritisch ist über die bislang

5. Evaluierung der Ergebnisse

nur sporadisch durchgeführte (zeitlich im Rahmen dieser Arbeit auch nicht intensiver zu betreibende) Qualitätssicherung zu urteilen. Insbesondere sind ausführliche Tests bei Einsätzen im Echtbetrieb notwendig, um die allgemeine Leistungsfähigkeit seriös bewerten zu können.

Eine klare Empfehlung kann für eine modellgetriebene Entwicklung unter Android ausgesprochen werden. Die in diesem Zusammenhang verwendeten Tools Xtext und Xtend erforderten zwar ein nicht unerhebliches Maß an Einarbeitungszeit, dafür leistet der daraus gewonnene Nutzen ein vielfaches an Mehrwert. Eine Anwendung macht allerdings nur auf weite Sicht überhaupt Sinn. Der Fokus muss bei einer modellgetriebenen Entwicklung auf der Wiederverwendbarkeit der entwickelten Strukturen liegen. Für einmalige Projekte ist der Aufwand, der in die Erarbeitung und Konzeption investiert werden muss, u. U. nicht gerechtfertigt.

6. Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde eine modellgetriebene Entwicklung eines Android SDKs bewerkstelligt. Ziel hierbei war es, die Schichten der Datenmodellierung/Datenhaltung und der Anbindung an ein Backend, die bei einer Appentwicklung anfallen, zusammenfassend zu kapseln, um dadurch Zeit für die eigentliche Appentwicklung zu gewinnen. Eine intuitive und einfache Nutzung ohne große Einarbeitungszeit waren weitere Ziele.

Das SDK sollte eine Anbindung an das von der Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt eigens entwickelte Orca-Backend ermöglichen. Hierbei handelt es sich allgemein betrachtet um eine schemafreie dokumentenorientierte Datenbank. Zusätzlich sollte auch eine Anbindung an die in dem Android-Betriebssystem standardmäßig enthaltene relationale SQLitedatenbank umgesetzt werden. Dabei soll der/die Entwickler/in jederzeit die Auswahl haben, welche der beiden Komponenten angesteuert werden soll. Es war sicherzustellen, dass beide "Welten" zueinander kompatibel sind und grundlegende Datenbankmethoden zur Verfügung stellen.

Die Entwicklung des SDKs sollte hierbei modellgetrieben erfolgen und aufzeigen, ob diese in einem Android-Kontext sinnvoll erscheint.

Zunächst wurde eine Domain-Specific Language (DSL) entworfen, die es ermöglicht, die Entitäten festzulegen, die im Kontext einer App von Bedeutung sind. Der Fokus hierbei lag auf einer vollständigen Abdeckung der Orca-Backend-Spezifikationen. Die festgelegten Entitäten werden durch einen im darauffolgenden Entwicklungsschritt implementierten Generator automatisch in Java-Klassen umgewandelt.

Der Generator wurde bewusst so konzipiert, dass möglichst wenig Code generiert werden sollte. Dadurch benötigt das SDK Plattformcode in Form von (überwiegend) generischen Klassen. Diese kapseln alle Orca-Backend- und SQLitedatenbank-Zugriffe. Das SDK besteht somit aus dem Zusammenspiel von generiertem Code und Plattformcode. Alle Funktionen des SDKs können nach einer erfolgten Generierung durch einfache Java-Methodenaufrufe genutzt werden.

Die im vorigen Kapitel durchgeführte Evaluierung kam zu dem Ergebnis, dass das entwickelte SDK das gesteckte Ziel einer Reduzierung der Einarbeitungszeiten erfüllt - und somit erheblich Zeit bei der Entwicklung einer App spart. Die wegfallende Einarbeitung

6. Zusammenfassung und Ausblick

in die durch das SDK gekapselten Schichten ermöglicht eine sofortige Implementierung des individuellen Appcodes.

Durch den gewählten Ansatz, den Plattformcode fast ausschließlich generisch zu implementieren, wird der generierte Code und erfreulicherweise auch der gesamte Codeumfang gering gehalten.

Die Ansteuerung aller Funktionen über Methodenaufrufe wurde als sehr positiv und intuitiv betrachtet. Alle Methoden kapseln vollständig den entsprechenden Bezugspunkt, sind selbsterklärend benannt und stellen zusätzlich per Java-Doc eine Dokumentation der jeweiligen Funktionen bereit. Die vollständige Kapselung der eigentlichen Komplexität birgt allerdings die Gefahr, dass es dadurch zu einer falschen bzw. zu einer häufigen Verwendung sehr aufwändiger Methoden kommt, woraus Fehler und Einschränkungen im Laufzeitverhalten einer App entstehen können.

Die Erfahrungen, die mit einer modellgetriebenen Entwicklung gesammelt wurden, sind durchwegs positiv und können für weitere Entwicklungen empfohlen werden. Hier muss allerdings klar abgewägt werden, ob sich der Aufwand lohnt. Im Rahmen dieser Arbeit wurde festgestellt, dass die Konzeption einer modellgetriebenen Entwicklung sehr viel Zeit in Anspruch nimmt. Einmalige Fallgestaltungen modellgetrieben zu entwickeln, ist demnach kaum empfehlenswert.

Das SDK konnte nur in sehr kleinem Rahmen und nur auf einer Hardwarekonfiguration getestet werden. Im Rahmen der Evaluation wurde deshalb eine bis dato unzureichende Qualitätssicherung ermittelt, die allerdings im Rahmen dieser Arbeit zeitlich auch nicht intensiver zu betreiben war.

Hier ist es wünschenswert, dass evtl. zukünftige Arbeiten an diesem Punkt fortsetzen. Denkbar wäre z.B. die Verwendung in Projektarbeiten oder als Teil einer Vorlesung zum Thema mobile Applikationen, um die Funktionsweise im Großen und die Kompatibilität zu verschiedenen Hardwarekonfigurationen überhaupt seriös beurteilen zu können. Insbesondere hierfür wurde unter Anhang C eine Schritt-für-Schritt-Anleitung zur Einrichtung des SDKs in eine Eclipse-Entwicklungsumgebung erstellt. Optimal wäre natürlich ein ausführliches und ausschließlich methodisches Testen über einen längeren Zeitraum.

Bezogen auf Android macht eine modellgetriebene Entwicklung z.B. im Bereich von Oberflächenelementen Sinn. Vorstellbar wäre hier die automatische Generierung von Activity- oder Fragment-Klassen. Ebenso denkbar wäre es betriebssystemübergreifend Code generieren zu lassen, damit App Portierungen beschleunigt werden können.

Abschließend soll hier noch erwähnt werden, dass für den Autor eine modellgetriebene Entwicklung vor der Erstellung dieser Arbeit absolutes Neuland war, fortgeschrittene Programmierkenntnisse in Bezug auf das Android-Betriebssystem waren dafür schon

6. Zusammenfassung und Ausblick

vorhanden. Durch die gewonnenen durchwegs positiven Erfahrungen und die Erweiterung vieler fachlicher Kenntnisse werden beide Bereiche auch künftig von hohem Interesse für den Autor bleiben.

A. Implementierte DSL mit Xtext

Listing A.1: Quellcode der entwickelten DSL (Quelle: Eigener Quellcode)

```
1 grammar org.xtext.orcasdk.entitymodel.EntityModel with
2   org.eclipse.xtext.common.Terminals
3
4 generate entityModel
5   "http://www.xtext.org/orcasdk/entitymodel/EntityModel"
6
7 Model:
8   appconstants=AppConstants?;
9
10 AppConstants:
11   'appconstants {'
12     'appname = ' valueappname+=(STRING)*; '
13     'apikey = ' valueapikey+=(STRING)*; '
14   '}'
15   packageentities+=Packages+
16 ;
17
18 Packages:
19   'package' name=QualifiedName '{'
20     entities+=AndroidEntity+
21   '}'
22 ;
23
24 QualifiedName:
25   ID ('.' ID)*
26 ;
27
28 AndroidEntity:
29   'entity' name = ID '{'
30     'TYPE = ' value+=(INT)+; '
31     attributes += AndroidAttribute+
32   '}'
33 ;
34 AndroidAttribute:
35   ((type='string' name=ID';') |
```

A. Implementierte DSL mit Xtext

```
35   (type='boolean' name=ID';') |  
36   (type='long' name=ID';') |  
37   (type='double' name=ID';') |  
38   (type='double[]' name=ID';') |  
39   (type='date' name=ID';') |  
40   (type='image' name=ID';') |  
41   (type='link' name=ID objectType=[AndroidEntity]  
42   {'linkproperties+=LinkProperties+' }))  
43 ;  
44  
45  
46 LinkProperties:  
47   ((type='string' name=ID';') | (type='boolean' name=ID';') |  
48   (type='long' name=ID';') | (type='double' name=ID';') |  
   (type='double[]' name=ID';') | (type='date' name=ID';'))  
49 ;
```

B. Inhalt des beigefügten Datenträgers

Der dieser Arbeit beigefügte Datenträger umfasst die zwei Ordner *Code* und *Thesis*. Im *Root*-Verzeichnis befindet sich diese Arbeit im PDF-Format in zwei Versionen. Eine ist für die Ansicht auf einem Bildschirm oder mobilem Endgerät gedacht (Endung “VIEW“). Die andere Version hat höhere Bildauflösungen und ist deshalb für einen Ausdruck geeignet (Endung “PRINT“).

Code

Dieser Ordner unterteilt sich in die Ordner *kompletter Code* und *Tutorial*. Letztgenannter wird für Anhang C benötigt. Unter erstgenanntem kann der vollständige Source-Code des für diese Arbeit entwickelten SDKs eingesehen werden. Dabei erfolgte eine Aufteilung auf die Ordner *Codegenerierung* und *Plattformcode*.

Thesis

In diesem Ordner befinden sich drei Unterordner:

- *Abbildungen*: Alle in dieser Arbeit abgedruckten Abbildungen sind hier in 600 dpi im png-Bildformat abrufbar.
- *Internetquellen*: Alle im Literatur- und Quellenverzeichnis angegebenen Quellen, die auf Seiten im Internet verweisen, befinden sich als PDF-Ausdruck in diesem Ordner.
- *Latex*: Hierin befinden sich alle *Latex*-Dateien, mit denen diese Arbeit kompiliert werden kann. Ebenso vorhanden ist das Literatur-Verzeichnis in Form einer *BibTeX*-Datei. Im Ordner *listings* sind alle abgedruckten Codebeispiele dieser Arbeit einsehbar. *pictures* beinhaltet die in der Arbeit verwendeten Abbildungen in einer niedrigeren Auflösung. Diese wurden zum Kompilieren der Arbeit mit dem Zusatz “VIEW“ verwendet.

C. Einrichtung des SDKs in eine Eclipse-Entwicklungsumgebung

Vorbemerkung

Diese Schritt-für-Schritt-Anleitung wurde nur auf einem Windows-System getestet. Dort kam es bei genauer Einhaltung der einzelnen Schritte zu keinerlei Fehler. Bei auftretenden Problemen bezüglich Xtext sind Kapitel 1 und 2 aus [3] oder die Xtext-Documentation unter [27] heranzuziehen.

Bei allen Abbildungen in diesem Kapitel handelt es sich um eigens erstellte Screenshots. Von einer individuellen Quellenangabe wird deshalb abgesehen.

WICHTIG: Die zu übernehmenden Xtext- und Xtend-Dateien sind nicht zu verändern! Es wird hier davon ausgegangen, dass vorab schon ein Java Development Kit (JDK) installiert wurde.

Einrichtungsschritte

1. Als Entwicklungsumgebung ist Eclipse zu verwenden. Die für diese Einweisung verwendete Version ist über die unter [4] genannte URL herunterladbar. Danach ist Eclipse zu starten.
2. Eclipse sind als nächstes die ADT-Tools für die Android-Entwicklung hinzuzufügen. Unter [9] ist unter dem Punkt “*Download the ADT Plugin*“ einsehbar, wie diese korrekt hinzuzufügen sind. Bei dem in Abbildung C.1 gezeigten Downloadfenster sind alle Tools auszuwählen und zu installieren.

C. Einrichtung des SDKs in eine Eclipse-Entwicklungsumgebung

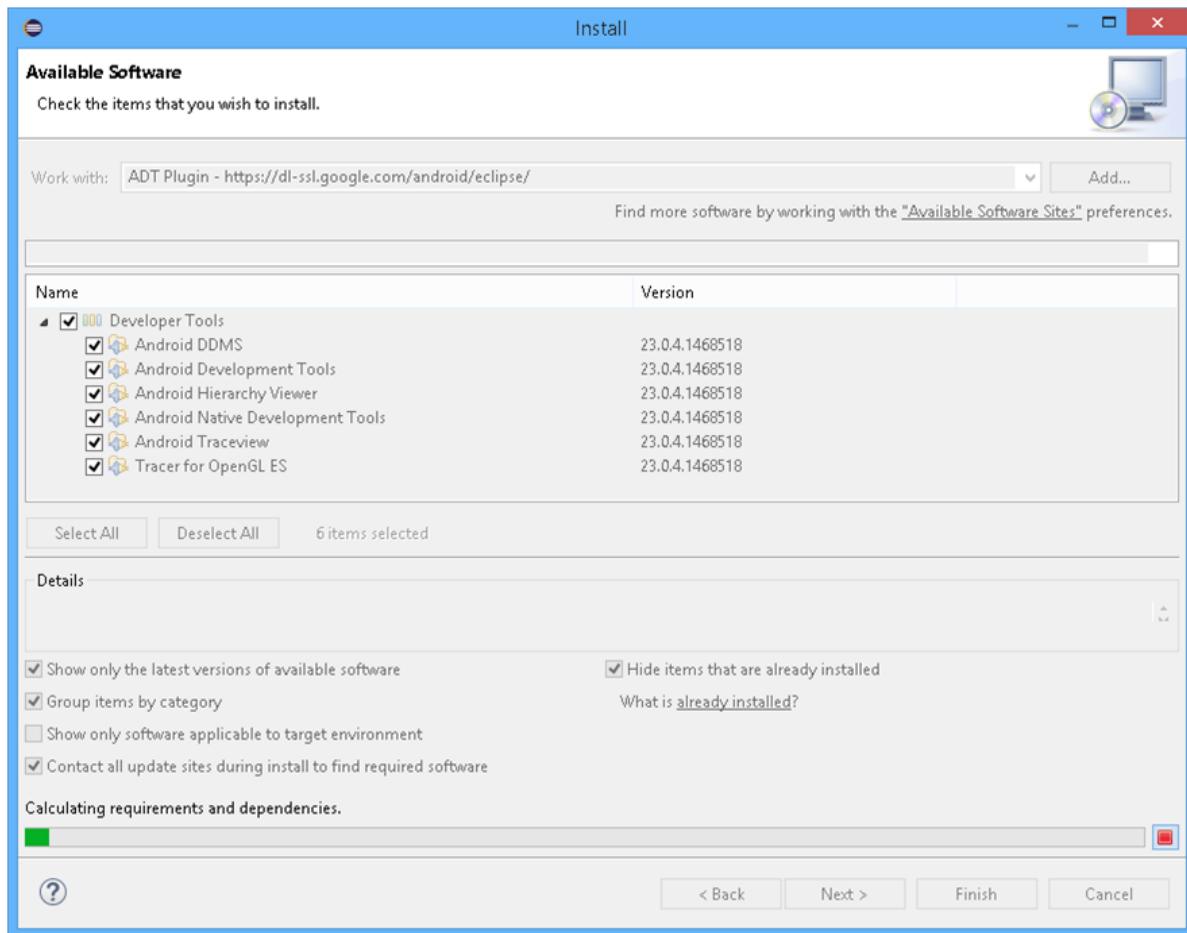


Abbildung C.1.: ADT zu Eclipse hinzufügen

3. Xtext (inklusive Xtend) ist analog wie die ADT-Tools über die Eclipse-Updatefunktion hinzuzufügen. Unter [28] ist der Link aufgeführt. Dort ist unter dem Punkt “*Update Sites*“ der Link unter “*Latest Release*“ zu kopieren und in der Updatefunktion unter Eclipse einzufügen. Auch hier (siehe Abbildung C.2) sind alle Tools zu installieren.

C. Einrichtung des SDKs in eine Eclipse-Entwicklungsumgebung

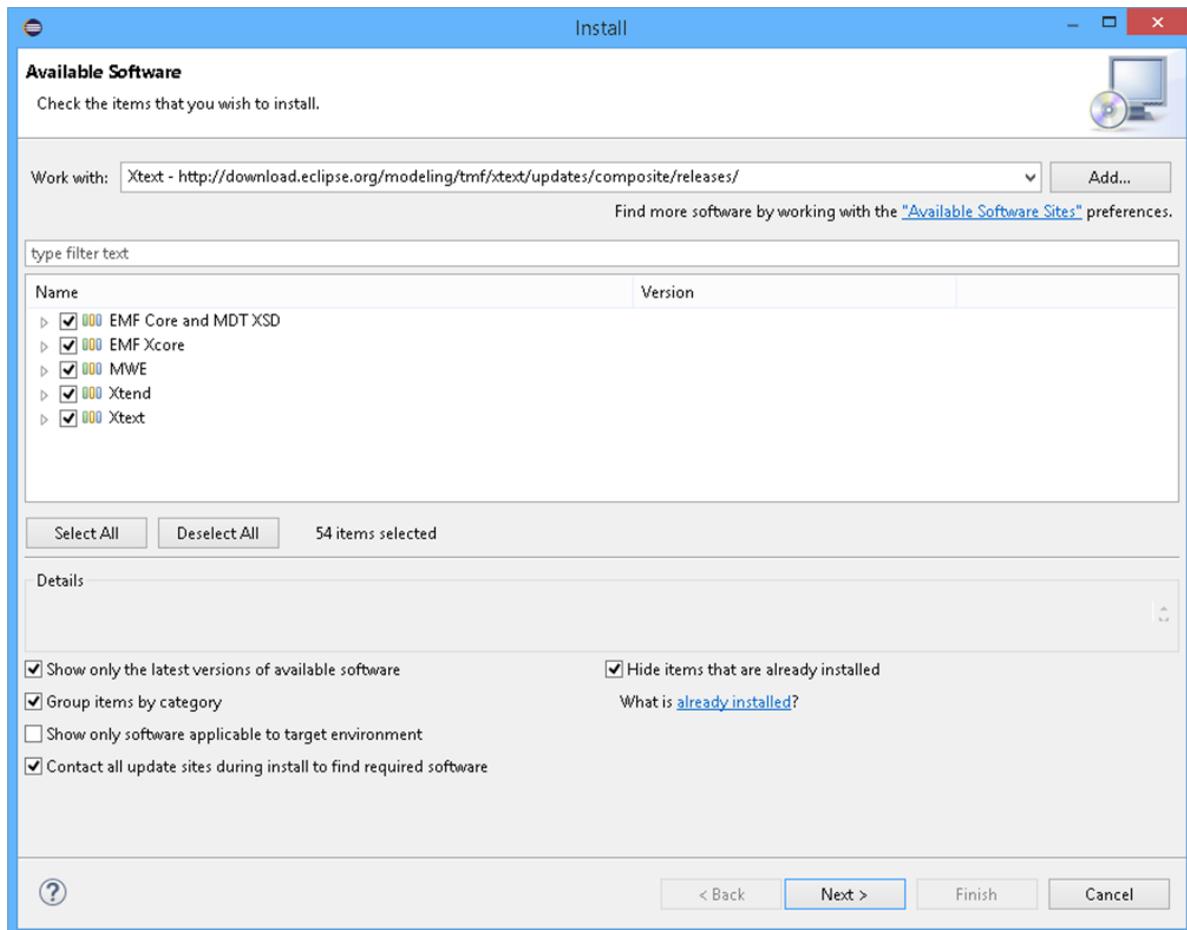


Abbildung C.2.: Xtext zu Eclipse hinzufügen

4. Wenn alles geklappt hat, können jetzt in Eclipse unter ->“File“ ->“New“ ->“Other“ die Punkte “Android“ und “Xtext“ ausgewählt werden.
5. Über den eben genannten Pfad ist als nächstes ein “Xtext Project“ anzulegen. Dort sind die Eingaben aus Abbildung C.3 identisch zu übernehmen.

C. Einrichtung des SDKs in eine Eclipse-Entwicklungsumgebung

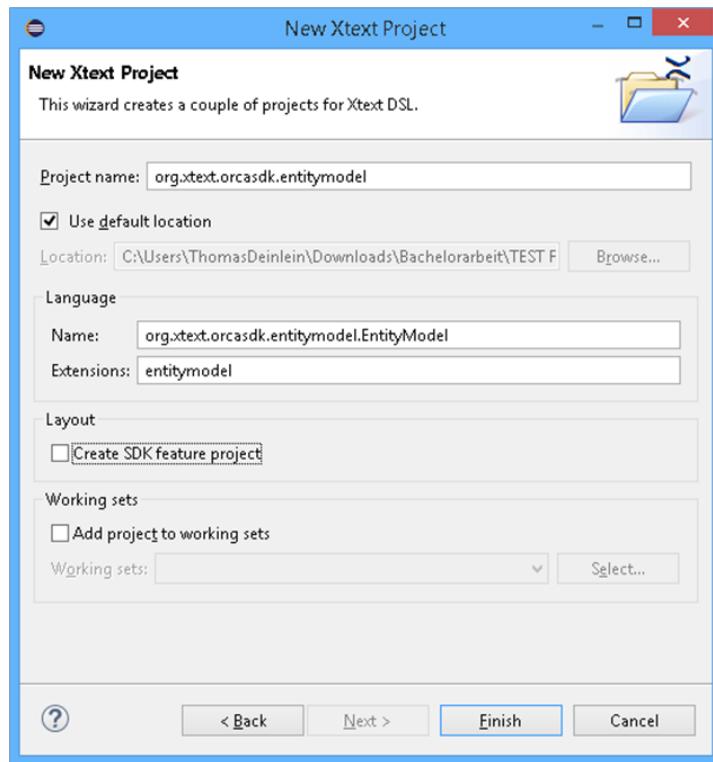


Abbildung C.3.: Erstellung Xtext Project

6. Nach dem Bestätigen durch “Finish“ erscheint Abbildung C.4.

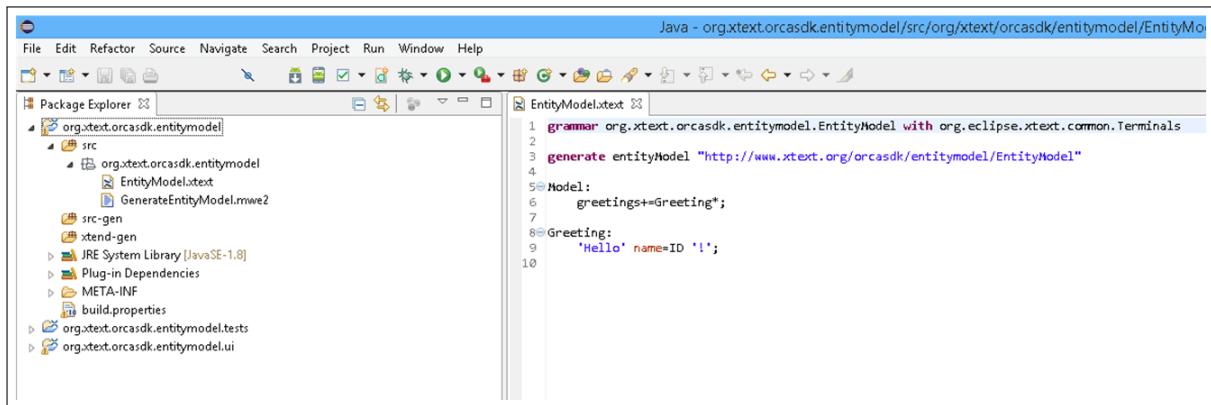


Abbildung C.4.: XtextProject

7. Die Dateien EntityModel.xtext und GenerateEntityModel.mwe2 sind aus dem Package org.xtext.orcasdk.entitymodel zu löschen (siehe auch hierzu Abbildung C.4).

C. Einrichtung des SDKs in eine Eclipse-Entwicklungsumgebung

8. Die soeben gelöschten Dateien sind durch Pendants von der dieser Arbeit beigefügten CD zu ersetzen. Die Dateien sind in dem Ordner Code\Tutorial\Codegenerierung\Xtext zu finden. Beide Dateien sind in das im zuvor genannten Punkt erwähnte Package des Xtext Projects zu kopieren.
9. Als nächstes sind folgende Schritte durchzuführen: Rechtsklick auf die Datei EntityModel.xtext, den Reiter “Run As“ auswählen und per Linksklick “1 Generate Xtext Artefacts“ bestätigen (siehe hierzu Abbildung C.5).

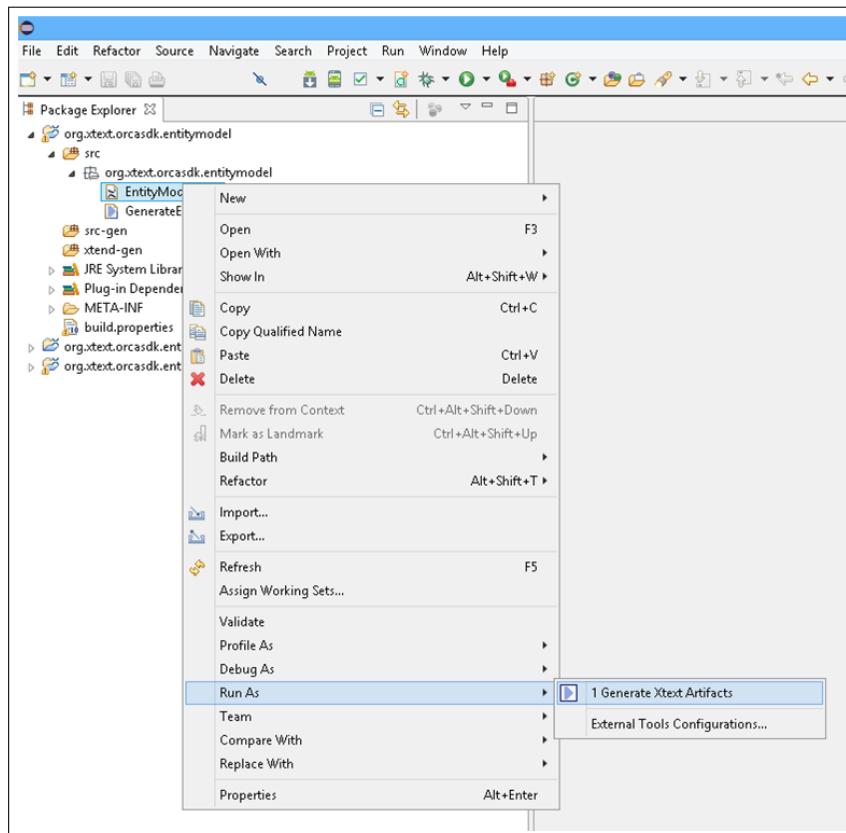


Abbildung C.5.: Generate Xtext Artefacts

10. Daraufhin ist der Download einer Datei notwendig. Die in Abbildung C.6 ersichtliche Abfrage ist mit Eingabe des Buchstabens “y“ und mit “enter“ zu bestätigen.

C. Einrichtung des SDKs in eine Eclipse-Entwicklungsumgebung

```

GenerateLanguageInfrastructure [org.xtext.resource.lib] [Mwe2Launch] [C:\Program Files\Java\java-8u21\bin] [www.cs (25.02.2015 17:09:54)
0 [main] WARN pes.access.impl.DeclaredTypeFactory - -----xtext.common.types-----
1 [main] WARN pes.access.impl.DeclaredTypeFactory - ASH library is not available. Falling back to java.lang.reflect API.
1 [main] WARN pes.access.impl.DeclaredTypeFactory - Please note that no information about compile time constants is available.
1 [main] WARN pes.access.impl.DeclaredTypeFactory - It's recommended to use org.objectweb.asm 5.0.1 or better.
1 [main] WARN pes.access.impl.DeclaredTypeFactory - -----
58 [main] INFO lipse.emf.mwe.utils.StandaloneSetup - Registering platform uri 'C:/Users/ThomasOenlein/Downloads/Bachelorarbeit/TEST FÜR VALIDIERUNG/myfirstapp'
65 [main] INFO lipse.emf.mwe.utils.StandaloneSetup - Adding generated EPackage 'org.eclipse.xtext.xbase.XbasePackage'
83 [main] INFO eclipse.emf.mwe.utils.GenModelHelper - Registered GenModel 'http://www.eclipse.org/Xtext/Xbase/XAnnotations' from 'platform:/resource/org.eclipse.xtext.xbase/model/Xbase.genmodel'
83 [main] INFO eclipse.emf.mwe.utils.GenModelHelper - Registered GenModel 'http://www.eclipse.org/xtext/xbase/types' from 'platform:/resource/org.eclipse.xtext.xbase/model/Xbase.genmodel'
846 [main] INFO eclipse.emf.mwe.utils.GenModelHelper - Registered GenModel 'http://www.eclipse.org/xtext/base/Xbase' from 'platform:/resource/org.eclipse.xtext.xbase/model/Xbase.genmodel'
846 [main] INFO eclipse.emf.mwe.utils.GenModelHelper - Registered GenModel 'http://www.eclipse.org/xtext/common/JavaVNTypes' from 'platform:/resource/org.eclipse.common.types/model/JavaVNTypes.e
1204 [main] INFO lipse.emf.mwe.utils.StandaloneSetup - Adding generated EPackage 'org.eclipse.xtext.common.types.typesPackage'

*ATTENTION*
It is recommended to use the ANTLR 3 parser generator (BSD licence - http://www.antlr.org/license.html).
Do you agree to download it (size 1MB) from 'http://download.itemis.com/antlr-generator-3.2.0-patch.jar'? (type 'y' or 'n' and hit enter)

```

Abbildung C.6.: Sicherheitsabfrage beim Ausführen von Xtext

- Nach erfolgreicher Ausführung müsste das Projekt aussehen wie in Abbildung C.7 abgebildet.

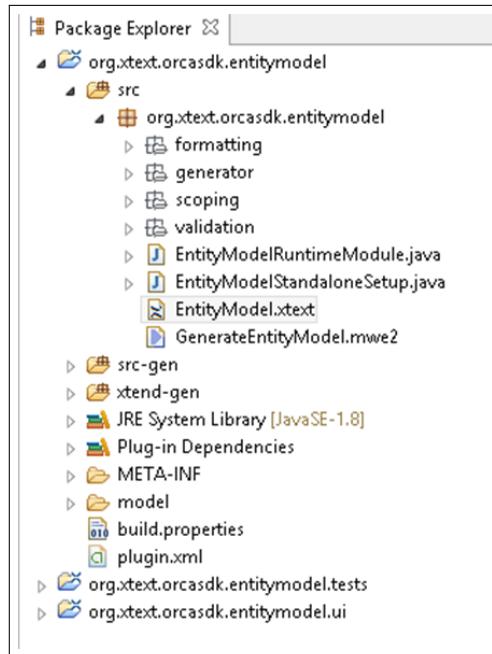


Abbildung C.7.: Veränderte Projektstruktur nach erfolgreicher Xtext-Ausführung

- Anschließend ist auf der Datei `GenerateEntityModel.mwe2` auch über einen Rechtsklick "Run As" -> "1 MWE2 Workflow" auszuführen (siehe Abbildung C.8).

C. Einrichtung des SDKs in eine Eclipse-Entwicklungsumgebung

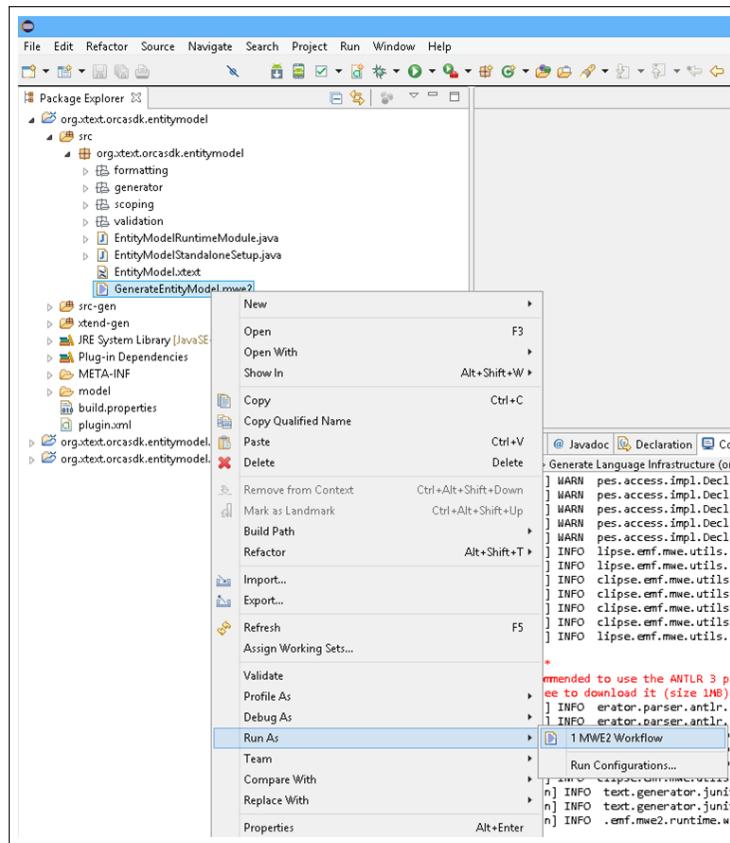


Abbildung C.8.: GenerateEntityModel.mwe2

13. In dem Package generator sind alle befindlichen Dateien zu löschen (nicht den Ordner).

Von der CD sind alle Dateien und Ordner aus dem Verzeichnis `Code\Tutorial\Codegenerierung\Xtend\` in dieses Package zu kopieren (siehe Abbildung C.9).

C. Einrichtung des SDKs in eine Eclipse-Entwicklungsumgebung

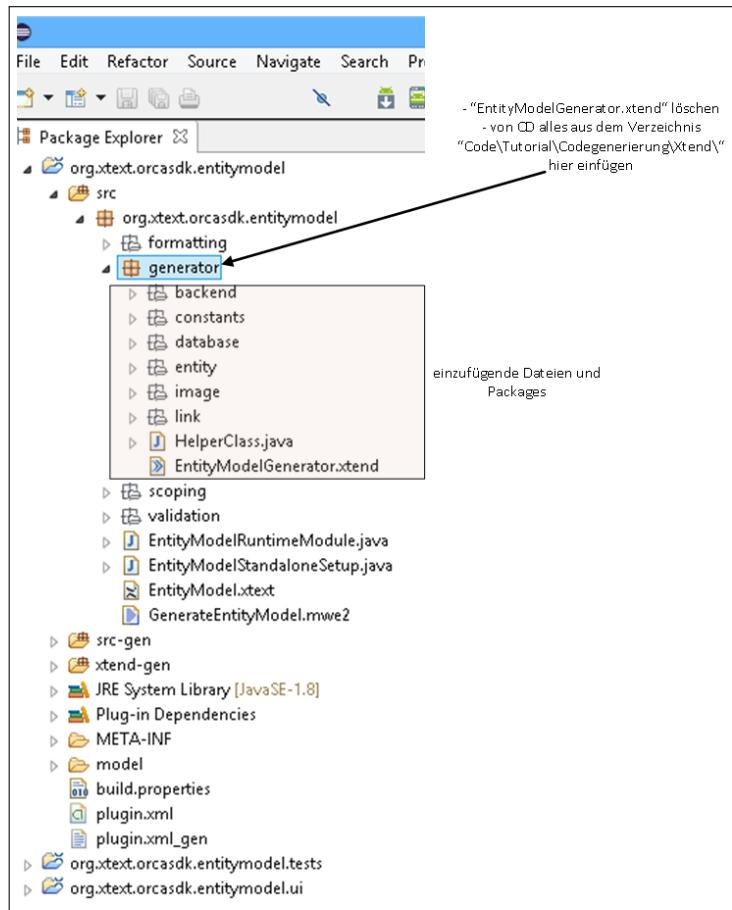


Abbildung C.9.: Hinzufügen aller Xtend-Klassen

14. Nun kann eine zweite Eclipse-Instanz gestartet werden, unter der das Domänenmodell festgelegt werden kann. Hierzu ist der “Run As“-Befehl auf das Package `org.xtext.orcasdk.entitymodel` anzuwenden (siehe Abbildung C.10).

C. Einrichtung des SDKs in eine Eclipse-Entwicklungsumgebung

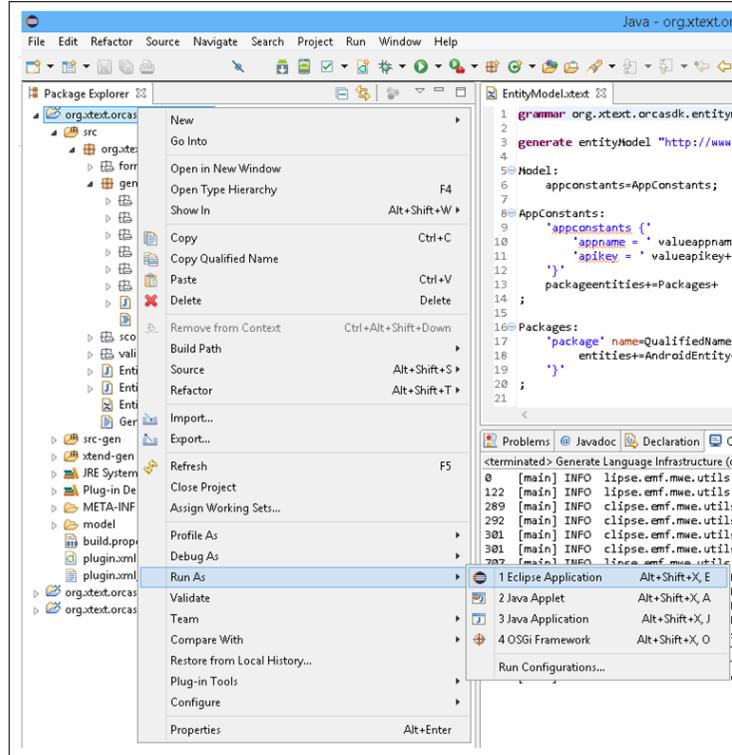


Abbildung C.10.: Starten der zweiten Eclipse-Instanz

15. Es öffnet sich eine neue Eclipse-Instanz. Dort ist über den “*Project Explorer*“ ein neues “*Android Application Project*“ zu erstellen. Hierbei ist anzumerken, dass alle im Rahmen dieser Arbeit durchgeföhrten Tests mit der API 18 (Android 4.3) durchgeföhr wurden.
16. Nachdem ein neues Android-Projekt angelegt wurde, ist dem *src*-Ordner per Rechtsklick eine Datei vom Typ “*File*“ (siehe Abbildung C.11) hinzuzufügen. Diese ist wie in Abbildung C.12 zu benennen. Die erscheinende Sicherheitsabfrage (siehe Abbildung C.13) ist zu bestätigen.

C. Einrichtung des SDKs in eine Eclipse-Entwicklungsumgebung

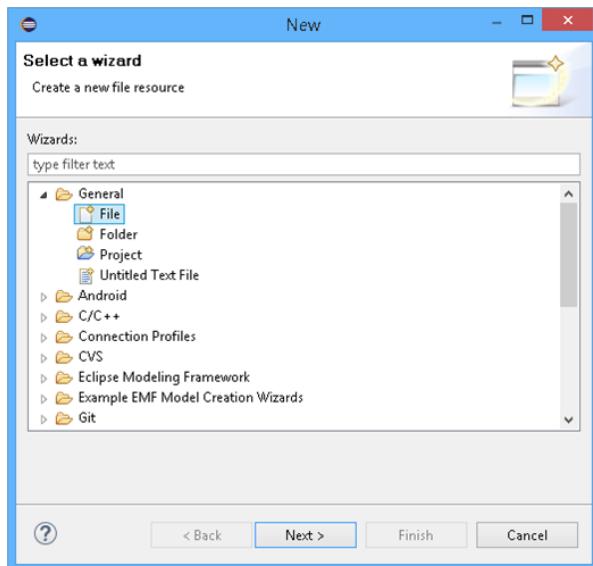


Abbildung C.11.: Erstellung einer “File“

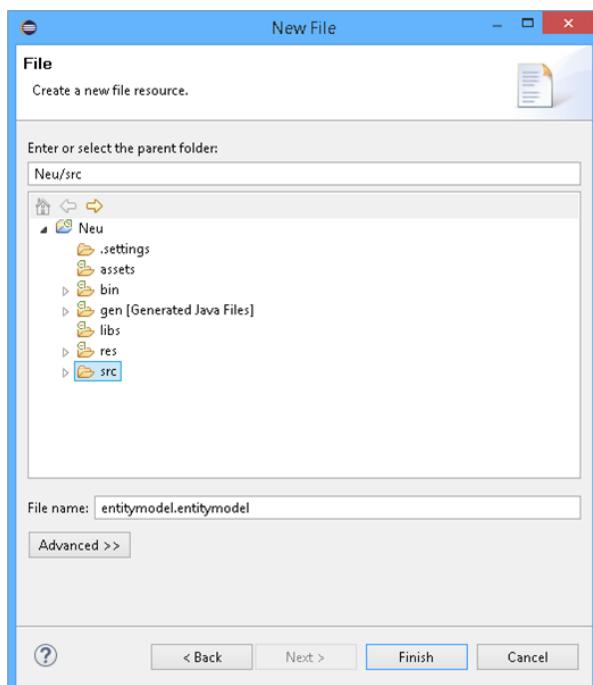


Abbildung C.12.: Benennung der “File“-Datei

C. Einrichtung des SDKs in eine Eclipse-Entwicklungsumgebung

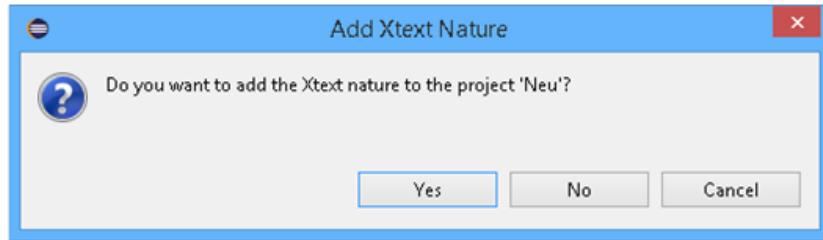


Abbildung C.13.: “Xtext nature“ zum Application Projekt hinzufügen

17. Abschließend ist jetzt noch der Plattformcode einzubinden. Im Ordner Code\Tutorial\Plattformcode\AndroidSDK\ sind drei JAR-Dateien enthalten, die alle(!) zum “build-Path“ des Android Projektes hinzugefügt werden müssen. Ein Kopieren der drei Dateien in den “libs“-Ordner war in der Testumgebung schon ausreichend, Eclipse erkannte die Dateien automatisch. Hat alles erfolgreich geklappt, werden in Eclipse keine Fehler mehr angezeigt.
18. Über das zuvor erstellte “File“ entitymodel.entitymodel kann nun das Domänenmodell festgelegt werden. Ein Beispiel ist in Abbildung C.14 zu sehen. Syntaxfehler werden analog Eclipse angezeigt.

The screenshot shows the Eclipse code editor with two tabs: "MainActivity.java" and "*entitymodel.entitymodel". The code in the editor is:

```
>MainActivity.java
*entitymodel.entitymodel

appconstants {
    appname = "deinName";
    apikey = "deinApiKey";
}

package de.meine.erste.app;
entity person{
    TYPE = 100;
    string name;
    string vorname;
    date geburtsdatum;
}
```

Abbildung C.14.: Festlegung eines Domänenmodells

19. Speichert man dieses über den entsprechenden Button ab, wird automatisch ein neuer Ordner mit Namen src-gen erstellt. Dieser muss zum build-path hinzugefügt werden, was in Abbildung C.15 veranschaulicht ist. In diesem Ordner befinden sich dann bereits die generierte Klasse.

C. Einrichtung des SDKs in eine Eclipse-Entwicklungsumgebung

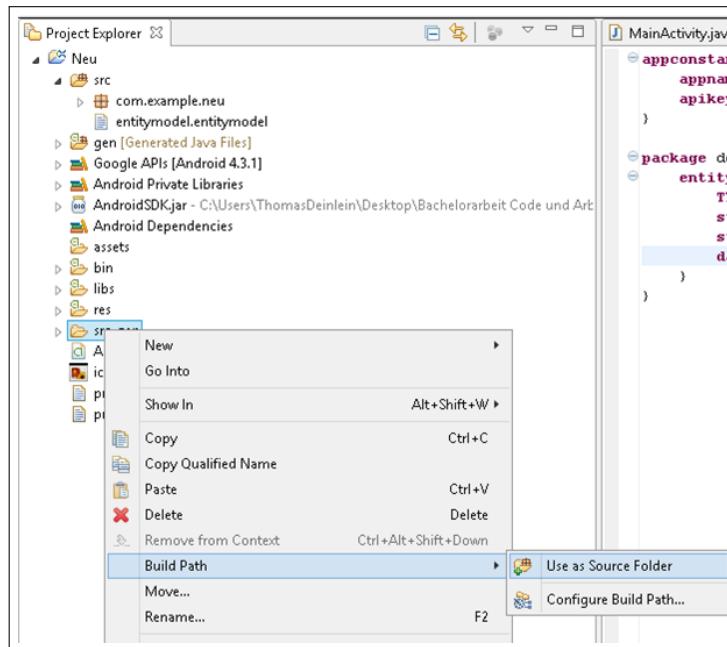


Abbildung C.15.: Hinzufügen des src-gen-Ordners zum build-path

20. Der src-gen-Ordner müsste nun wie in Abbildung C.16 aussehen.

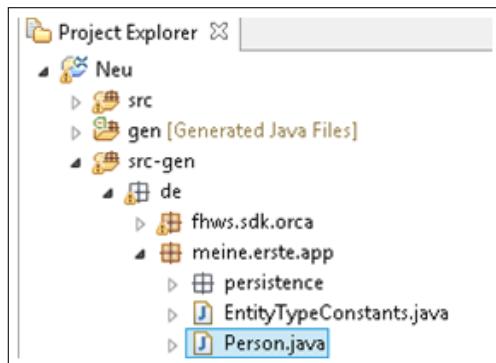


Abbildung C.16.: Generierte Klasse Person im src-gen-Ordner

21. Das entwickelte Android SDK ist nun in vollem Umfang verfügbar. Der Klasse Person stehen alle drei Schnittstellen von nun an zur Verfügung (siehe Abbildung C.17). Bei jeder Änderung des Domänenmodells und nachfolgender Speicherung erfolgt eine neue Generierung der entsprechenden Klassen.

C. Einrichtung des SDKs in eine Eclipse-Entwicklungsumgebung

NUR_BACKEND: Methode zum Absetzen einer SQL-Query ans Orca-Backend. In der Response wird eine EntityPage mit den gefundenen Entities zurückgesendet. BACKEND-ENDPUNKT: /api/{appname}/entities

Parameters:

- queryString der SQL-Query-String
- orderBy ein OrderBy als String, es kann auch null übergeben werden
- offset das Offset für die EntityPage
- size die Anzahl an Entities, die die EntityPage maximal enthalten soll
- callback Callback über das die Ergebnis-entityPage, der HttpStatusCode und eine errorMessage abgerufen werden können

Abbildung C.17.: Person verfügt über alle drei Schnittstellen

Anmerkung: Damit Zugriffe zum Orca-Backend durchgeführt werden können, muss in der entsprechenden AndroidManifest.xml die Permission uses-permission android:name=\android.permission.INTERNET\ hinzugefügt sein!

Abbildungsverzeichnis

1.1.	Marktanteil mobiler Betriebssysteme 2013	2
1.2.	Anzahl Apps in App-Stores	3
1.3.	Beispielhaftes Schichtenmodell zwischen Frontend und Backend	4
2.1.	Zusammenspiel von DSL, Domänenmodell und Generator	12
2.2.	Systemarchitektur von Android	13
2.3.	Vergleich von JVM und DVM	14
2.4.	Datenfluss einer AsyncTask-Klasse	17
2.5.	Funktionsweise des Orca-Backend ohne das SDK	21
3.1.	Spezifikation der Orca-Backend-Schnittstelle	29
3.2.	Spezifikation der SQLitedatenbank-Schnittstelle	34
3.3.	Spezifikation der kombinierten Schnittstelle	39
4.1.	UML-Klassendiagramm Entity-Workflow	46
4.2.	Wrapperklassen vereinfachen die Verwendung generischer Klassen	52
4.3.	Beim erstmaligen Speichern angelegte Tabellen	57
4.4.	Tatsächlicher Workflow der kombinierten Schnittstelle	61
5.1.	Auswahl einer Methode in Eclipse	68
C.1.	ADT zu Eclipse hinzufügen	81
C.2.	Xtext zu Eclipse hinzufügen	82
C.3.	Erstellung Xtext Project	83
C.4.	XtextProject	83
C.5.	Generate Xtext Artefacts	84
C.6.	Sicherheitsabfrage beim Ausführen von Xtext	85
C.7.	Veränderte Projektstruktur nach erfolgreicher Xtext-Ausführung	85
C.8.	GenerateEntityModel.mwe2	86
C.9.	Hinzufügen aller Xtend-Klassen	87
C.10.	Starten der zweiten Eclipse-Instanz	88
C.11.	Erstellung einer “File“	89
C.12.	Benennung der “File“-Datei	89
C.13.	“Xtext nature“ zum Application Projekt hinzufügen	90
C.14.	Festlegung eines Domänenmodells	90
C.15.	Hinzufügen des src-gen-Ordners zum build-path	91

Abbildungsverzeichnis

C.16.Generierte Klasse Person im src-gen-Ordner	91
C.17.Person verfügt über alle drei Schnittstellen	92

Tabellenverzeichnis

5.1.	Vergleich des zeitlichen Aufwands vor Schreiben des Appcodes	64
5.2.	Umfang des Plattformcodes bei minimaler Codegenerierung	65
5.3.	Umfang des generierten Codes für Entität Person bei minimaler Codegenerierung	65
5.4.	Umfang des Plattformcodes bei minimalem Plattformcode	66
5.5.	Umfang des generierten Codes für Entität Person bei minimalem Plattformcode	66
5.6.	Gegenüberstellung des jeweiligen gesamten Codeumfangs	67

Listings

2.1. Beispiel-DSL mit <i>Xtext</i>	9
2.2. Festlegung eines Domänenmodells anhand der Beispiel-DSL	10
2.3. Codegenerator mit <i>Xtend</i>	11
2.4. Beispiel-Template in <i>Xtend</i>	11
2.5. Beispiel für ein JSON-Entity	22
4.1. Beispiel Domänenmodell	43
4.2. Ausschnitt <i>Xtend</i> -Generator	44
4.3. AppConstants- <i>Xtend</i> -Template	45
4.4. Konstruktor einer generierten Klasse Person	48
4.5. Methode der Orca-Backend-Schnittstelle mit Callback	53
5.1. Beispiel Domänenmodell	65
5.2. Eine Methode kapselt Backend und Datenbank vollständig	69
5.3. Fehler bei nachfolgendem Methodenaufruf	69
5.4. Korrekte Handhabung aufeinander aufbauender Methodenaufrufe	70
A.1. Quellcode der entwickelten DSL	77

Literatur- und Quellenverzeichnis

- [1] Ralf Adams. *SQL: Eine Einführung mit vertiefenden Exkursen*. München: Hanser, 2012. ISBN: 9783446432789.
- [2] Arno Becker und Marcus Pant. *Android 4.4: Programmieren für Smartphones und Tablets - Grundlagen und fortgeschrittene Techniken*. 3rd ed. Heidelberg: dpunkt.verlag, 2014. ISBN: 978-3-86491-458-4.
- [3] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend: Learn how to implement a DSL with Xtext and Xtend using easy-to-understand examples and best practices*. Community experience distilled. Birmingham, UK: Packt Pub., 2013. ISBN: 9781782160304.
- [4] Eclipse Entwicklungsumgebung. URL: <http://www.eclipse.org/downloads/> (besucht am 02.03.2015).
- [5] *Genson Framework, User Guide*. URL: <http://owlike.github.io/genson/Documentation/UserGuide/> (besucht am 02.03.2015).
- [6] Google, Hrsg. *Android Developer: Processes and Threads*. URL: <http://developer.android.com/guide/components/processes-and-threads.html> (besucht am 02.03.2015).
- [7] Google Inc., Hrsg. *Android-Runtime*. URL: <https://source.android.com/devices/tech/dalvik/index.html> (besucht am 02.03.2015).
- [8] Michael Inden. *Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung*. 2., akt. u. erw. Aufl. Heidelberg, Neckar: dpunkt, 2012. ISBN: 9783864900051.
- [9] *Installing the Android Eclipse Plugin*. URL: <http://developer.android.com/sdk/installing/installing-adt.html> (besucht am 02.03.2015).
- [10] Christoph Kecher. *UML 2.3: Das umfassende Handbuch*. 4., erw. Ausg. Galileo Computing. Bonn: Galileo Press, 2011. ISBN: 9783836217521.
- [11] Dirk Louis und Peter Müller. *Android: Der schnelle und einfache Einstieg in die Programmierung und Entwicklungsumgebung ; [auf DVD: Beispiele, Tutorials, JRE und Android-Bundle ; inkl.: Java-Tutorium für Ein- und Umsteiger]*. München: Hanser, 2014. ISBN: 9783446438316.
- [12] Mark Massé. *REST API design rulebook*. 1st ed. Sebastopol, CA: O'Reilly, 2012. ISBN: 9781449310509.

Literatur- und Quellenverzeichnis

- [13] Zigurd Mednieks. *Android-Programmierung: [Java-Programmierung für moderne mobile Endgeräte]*. 2. Aufl. Beijing [u.a.]: O'Reilly, 2013. ISBN: 9783955611415.
- [14] *Orca-Backend Datenstrukturen*. URL: <http://applab.fhws.de/projects/orca/json-chapter3.html> (besucht am 02.03.2015).
- [15] *Orca-Backend, Entitäten-Endpunkte*. URL: <http://applab.fhws.de/projects/orca/entity-endpoints.html> (besucht am 02.03.2015).
- [16] *Orca-Backend Überblick*. URL: <http://applab.fhws.de/projects/orca/overview.html> (besucht am 02.03.2015).
- [17] R.T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation*. 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf (besucht am 02.03.2015).
- [18] *Schichten einer App mit Backend-Anbindung*. URL: <http://t3n.de/news/backend-service-perfekt-mobile-483451/> (besucht am 02.03.2015).
- [19] Thomas Stahl. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. 2., aktualisierte und erw. Aufl. Heidelberg: Dpunkt-Verl., 2007. ISBN: 3898644480.
- [20] statista, Hrsg. *Anzahl der Apps in den Top App Stores*. URL: <http://de.statista.com/statistik/daten/studie/208599/umfrage/anzahl-der-apps-in-den-top-app-stores/> (besucht am 02.03.2015).
- [21] statista, Hrsg. *Marktanteil bei Smartphones nach Betriebssystem weltweit seit 2009 bis 2013 (Ausschnitt)*. URL: <http://de.statista.com/statistik/daten/studie/12885/umfrage/marktanteil-bei-smartphones-nach-betriebssystem-weltweit-seit-2009/> (besucht am 02.03.2015).
- [22] statista, Hrsg. *Prognose zum Smart Home Umsatz weltweit*. URL: <http://de.statista.com/statistik/daten/studie/318059/umfrage/prognose-zum-smart-home-umsatz-weltweit/> (besucht am 02.03.2015).
- [23] statista, Hrsg. *Prognose zum Umsatz mit Wearable-Computing-Geräten in Europa*. URL: <http://de.statista.com/statistik/daten/studie/322222/umfrage/prognose-zum-umsatz-mit-wearable-computing-geraeten-in-europa/> (besucht am 02.03.2015).
- [24] statista, Hrsg. *Prognose zur Anzahl der Smartphone-Nutzer weltweit von 2012 bis 2018 (in Milliarden)*. URL: <http://de.statista.com/statistik/daten/studie/309656/umfrage/prognose-zur-anzahl-der-smartphone-nutzer-weltweit/> (besucht am 02.03.2015).
- [25] Stefan Tilkov. *REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien*. dpunkt.verlag, 2011. ISBN: 9783898648660.
- [26] *Xtend-Templates, Xtend-Documentation*. URL: <http://eclipse.org/xtend/documentation.html#templates> (besucht am 02.03.2015).

Literatur- und Quellenverzeichnis

- [27] *Xtext Documentation*. URL: <http://eclipse.org/Xtext/documentation/2.7.0/Xtext%20Documentation.pdf> (besucht am 02.03.2015).
- [28] Xtext Download-Link. URL: <https://eclipse.org/Xtext/download.html> (besucht am 02.03.2015).

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.

Thomas Deinlein, am 6. März 2015