Bakkalaureatsarbeit

Android-Programmierung

Erstellung einer Android-Applikation für die Job- und Karrieremesse Connect

Michael Wutti

Alpen-Adria Universität Klagenfurt

2014 bis 2016

Inhaltsverzeichnis

1. Einleitung

Im Rahmen eines Softwarepraktikums an der Alpen-Adria Universität Klagenfurt, wurde eine Android-App zur Job- und Karrieremesse Connect erstellt. Dieses Softwarepraktikum ist im Sommer 2014 per E-Mail ausgeschrieben worden. Die nachfolgende Bakkalaureatsarbeit umfasst sowohl eine inhaltliche, als auch eine technische Dokumentation, des Entwicklungsprozesses dieser Android-App.

Das Ziel des Projektes war es, eine kostenfreie Android-App zur Verfügung zu stellen, mit der die Besucher/innen digital und mobil, alle aktuellen Informationen zur Messe abrufen können. Die Messe fand am Donnerstag den 13. November 2014 von 09:00–15:00 Uhr in der Alpen-Adria Universität Klagenfurt statt.

Der Zeitrahmen, bis zur Fertigstellung der App, war bis spätestens 13. November 2014 festgelegt. Der Folder für die Besucher/innen, auf dem ein QR-Code der App aufgedruckt werden sollte, erschien bereits einen Monat früher. Somit sollte eine finalisierte Version der App bereits zu diesem Zeitpunkt vorhanden und funktionsfähig sein.

Bei der Entwicklung der App waren folgende Personen beteiligt:

- **Herr Assoc.Prof. Dipl.-Ing. Dr. Klaus Schöffmann**
  Begleiter des Softwarepraktikums sowie Ansprechperson bei programmier- und entwicklungstechnischen Aspekten.

- **Herr Dipl.-Ing. Dr. Bonifaz Kaufmann**
  Technischer Begleiter und Ansprechperson im Bereich Android-Programmierung.

- **Frau Mag.Phil. Johanna Ortner**
  Inhaltlicher Support und organisatorische Verantwortung.

Die Entwicklungszeit der App umfasste bis zur Veröffentlichung im *Google Play Store*[1] fünf Monate. Bis zum Messetermin am 13. November wurden weiterhin Verbesserungen und inhaltliche Änderungen vorgenommen, sodass sich der gesamte Zeitrahmen auf sechs Monate ausdehnte. Die erste Version der App wurde am 16. Oktober 2014 in den *Play Store* hochgeladen.

Einer der zentralen Gedanken bei der Erstellung der App war es, diese mehrere Jahre verwenden zu können. So wurden für das Jahr 2015 die Inhalte aktualisiert und die App konnte erfolgreich wiederverwendet werden.

1.1 Connect: Die Job- und Karrieremesse in Kärnten

Zahlreiche nationale und internationale Unternehmen und Bildungseinrichtungen präsentieren sich auf der größten Job- und Karrieremesse in Kärnten. Sie bietet Studierenden, Absolvent/innen, Maturant/innen und jobinteressierten Personen die Möglichkeit, sich über berufliche Chancen am Arbeitsmarkt zu informieren und bereits erste Kontakte zu Unternehmen und Bildungseinrichtungen zu knüpfen.

Zum einen erhalten angehende Arbeitnehmer/innen Informationen über Arbeitsmarkt, Berufswahl, Studienwahl, Zukunftsplanung und Studieren im Ausland, zum anderen wird jungen Startup-Unternehmen, bis hin zu international anerkannten Konzernen, die Möglichkeit geboten, ihre zukünftigen Mitarbeiter/innen persönlich kennenzulernen und auf Anforderungen im Arbeitsmarkt vorzubereiten.

Den Besucher/innen der Messe wird darüber hinaus ein umfangreiches Rahmenprogramm geboten. Impulsvorträge, Diskussionen, Expertentipps zur erfolgreichen Weiterbildung und den neuesten Entwicklungen am Arbeitsmarkt, sind neben geplanten Messespecials und Unterhaltungsevents, weitere Highlights der Messe.

1.2 Praktikum zur Bakkalaureatsarbeit

Im Sommer 2014 wurde von der Alpen-Adria Universität Klagenfurt, an alle Studierende der Fakultät für technische Wissenschaften, ein Praktikum zur Bakkalaureatsarbeit ausgeschrieben. Ziel sollte es sein, eine Messe-App zu planen und zu erstellen, die den Besucher/innen bereits vor dem Messebeginn zur Verfügung steht.

Die Inhalte und der Funktionsumfang wurden folgend beschrieben:

- Startscreen mit dem Connect-2014 Logo, Datum und Uhrzeit inklusive Logos der großen Partner der Messe
- Liste aller Aussteller, inklusive Standnummer und eventueller Verlinkung zu den Unternehmenshomepages (Unternehmensprofile zum Download)
- Lageplan verbunden mit der Austellerliste inklusive Standnummer etc.
- Liste aller Messespecials inklusive Standnummern, verbunden mit dem Lageplan und/oder der Ausstellerliste
- Impulsvortragsprogramm

1.3 Planung des Entwicklungsprozesses

Der erste Schritt in der Entwicklung der App bestand darin, mit den involvierten Personen Kontakt aufzunehmen. Die Inhalte und Funktionen der App, sowie Rahmenbedingungen und organisatorische Fragen wurden besprochen und es wurde ein erster grober Ablauf des Projektes erarbeitet.

Dieser Ablauf unterteile sich in drei größere Entwicklungsphasen. Der erste Schritt umfasste die Anforderungsanalyse, Planung und Gestaltung des User Interface (*UI*) und die Erstellung und Implementierung eines Prototyps. In der zweiten und längsten Phase wurde die Funktionsweise und Architektur der App geplant und umgesetzt. Ein Datenbankmodell wurde erstellt und die Funktionen der App wurden mit Hilfe von Testdaten implementiert. Im dritten Abschnitt sind die Produktivdaten in die App migriert worden und die letzten Änderungen wurden durchgeführt.

Im nachfolgenden Abschnitt wird die erste Phase (Anforderungsanalyse und Prototyping) des Entwicklungsprozesses vorgestellt und dokumentiert.

Abb. 1 Besucherfolder 2013

2. Prototyping

Ein weitverbreiteter Ansatz in der Softwareentwicklung, vor allem in der Entwicklung von mobilen Applikationen, ist die Erstellung eines Prototyps. Ein Prototyp ist ein Modell, eines oder mehrerer Teilsysteme einer Applikation und es wird unter anderem das *UI* in abstrakter oder konkreter Form dargestellt. Dieser Prototyp wird anschließend, als Basis für die Entwicklung der Software, im gesamten Entwicklungsprozess verwendet. Er vereinfacht die Kommunikation mit Kunden, aber auch die Prozesse im eigenen Entwicklungsteam. Es können Lösungsansätze erarbeitet werden, ohne bereits eine bestehende Implementierung eines Systems zu besitzen. Daraus ergibt sich die Möglichkeit, Probleme frühzeitig zu erkennen und Änderungswünsche eines Kunden mit wenig Aufwand durchzuführen, bevor die Software mit hohem Kostenaufwand geändert werden muss.

2.1 Arten und Eigenschaften von Prototypen

Prototypen können in verschiedenster Art und Weise realisiert werden. Angefangen von einem per Hand erstellten Papierprototypen, über softwaregestützte Prototypingwerkzeuge, bis hin zu Prototypen, die bereits teilautomatisierten Programmcode erzeugen.

In der Softwareentwicklung wird unter anderem zwischen folgenden grundlegenden Arten von Prototyping unterschieden[2].

- **Exploratives Prototyping**
  Ziel ist es, eine Anforderungsspezifikation zu erstellen und darauf aufbauend die Problemlösungsstrategien zu beurteilen und zu verifizieren. Hierbei wird der Fokus vor allem auf die Funktionalität der Applikation gesetzt. Als Ergebnis sollte festgestellt werden, ob die Spezifikationen und Funktionen der Applikation die Anforderungen erfüllen.

- **Evolutionäres Prototyping**
  Hierbei wird über den gesamten Softwareentwicklungsprozess ein aktueller und funktionsfähiger Prototyp gewartet. Der Prototyp wird initial mit den wichtigsten Funktionalitäten ausgestattet und nach und nach verbessert und erweitert. Bei dieser Methode stehen vor allem der ständige Kundenkontakt und das daraus erzielte Feedback im Vordergrund.

- **Experimentelles Prototyping**
  Der Prototyp wird für das Finden von Problemlösungen genutzt. Es können verschiedenste Lösungsansätze angewendet und somit Erfahrungen und Erkenntnisse gesammelt werden. Hier steht vor allem die Analyse und Forschung im Vordergrund.

Darüber hinaus existieren noch weitere Ansätze wie beispielsweise *horizontales* und *vertikales* Prototyping, bei denen nur gewisse Teile und Komponenten einer Anwendung modelliert werden.

2.2 Vor- und Nachteile des Prototypings

Prototyping im frühen Entwicklungsstadium einer Applikation und während das gesamten Entwicklungsprozess, eröffnet viele Möglichkeiten, welche Erfolgschancen und die Qualität der Applikation erheblich steigern können. Es zieht jedoch nicht nur Vorteile, sondern auch Nachteile mit sich.

- **Vorteile**
  - Durch ständigen Kundenkontakt und Evaluierung, sinkt das Risiko einer Entwicklung in eine Richtung, die der Kunde gar nicht wünscht oder nicht die Ziele und Anforderungen erfüllt.
  - Änderungen am System können im Prototyp kostengünstig und schnell vorgenommen werden, während hingegen an einer lauffähigen Applikation wesentlich mehr Aufwand entstehen würde.
  - Probleme und Hindernisse können frühzeitig erkannt werden.
  - Eine abstrakte Darstellung des Systems bietet mehr Freiraum für neue Ideen.
  - Es entstehen weniger Korrekturarbeiten am Ende des Entwicklungsprozesses.

- **Nachteile**
  - Durch Prototyping entstehen zusätzliche Kosten
  - Durch zu schnelles Prototyping besteht oft die Gefahr, Anforderungen zu ungenau zu erfassen und somit den Entwicklungsprozess zu erschweren

2.3 Connect-App Prototyp

Abb. 2 Prototyping

Die Prototypingmethode für die Connect-App entspricht am ehesten dem Konzept des *explorativen Prototypings.* Es wurde zuerst ein Prototyp auf Papier entwickelt, der die Anforderungen der Applikation modelliert. Im Zuge dieses Prozesses sind neue Ideen entstanden und bestehende Anforderungen stellten sich als unnötig heraus. Dieser Papierprototyp wurde anschließend digitalisiert, mit einem Sartphone-Mockup[3] versehen (Abb. 2) und als interaktive PDF-Präsentation aufgearbeitet, womit der Kontrollfluss der Applikation besser abgebildet werden konnte. Der Abstraktionslevel blieb somit sehr hoch und gleichzeitig entstand ein hochwertiger digitalisierter Prototyp.

Nachfolgend wird der Papierprototyp präsentiert und es werden die daraus erzielten Ergebnisse, in Bezug auf Anforderungsanalyse und Funktionalität erörtert. Gleichzeitig wird dem Prototypen die tatsächliche Applikation gegenübergestellt, um besser auf die erzielten Resultate des Prototypingprozesses eingehen zu können.

2.4 Startscreen

Abb. 3 Startscreen

Der Startscreen ist der zentrale Einstiegspunkt, der beim Öffnen der App angezeigt wird. Er soll den Benutzer/innen einen Überblick über Themenbereiche, Umfang und Funktionalität der Applikation liefern. Wie bereits in Abb. 2 angedeutet, wurde der Startscreen den Besucher/innen-Folder der Messe nachempfunden. Die Anforderungen in Abschnitt 1.2 wurden erfüllt und in kompakter Form dargestellt. Zusätzlich ist ein Link hinzugefügt worden, mit dem man über den Smartphonebrowser direkt zur Homepage der Karrieremesse gelangt.

Außerdem wird am Prototyp eine weitere Funktion angedeutet. In der rechten oberen Ecke befindet sich ein Button mit einer Lupe, was auf eine Suchfunktion hinweisen soll. Die Benutzer/innen können mit dieser Funktion, durch Eingabe eines Namens oder einer Standnummer, direkt zum entsprechenden Austellerprofil gelangen. Die Suchfunktion wurde jedoch in der produktiven Applikation in das Navigationsmenü ausgelagert.

In der linken oberen Ecke findet man den Button zum Aufrufen des Navigationsmenüs. Bei Android-Applikationen werden häufig drei übereinanderliegenden Balken verwendet, um eine Liste von Funktionen anzudeuten.

2.5 Navigationsmenü

Abb. 4 Navigationsmenü

Das Navigationsmenü wurde als *NavigationDrawer* (siehe Abschnitt 4.4) realisiert. Dies ist eine häufig verwendete Form der Navigation in modernen Android-Applikationen. Die Benutzer/innen können dadurch in jeder Ansicht zwischen den Funktionen navigieren. Der Aufruf geschieht entweder durch eine Berührung der linken oberen Ecke oder durch einen Wisch (*Swipe*) von der linken Kante des Smartphones nach rechts. Die *Swipe*-Geste ist besonders komfortabel, da die Benutzer/innen das Smartphone oft in einer Hand halten und der Button des Navigationsmenüs meist nicht ohne eine unangenehme Überstreckung des Daumens erreichbar ist. Weitere Informationen zum Navigationsmenü der App befinden sich in Abschnitt 4.

Im Prototyp wurden die Funktionen zusätzlich in die Bereiche *Programm* und *Orientierung* eingeteilt. Für die produktive Applikation wurde entschieden, auf diese Gliederung zu verzichten. Trotzdem hätte es eine Erleichterung für die Benutzer/innen gewesen sein können, die acht Listenelemente, etwa mit Abständen zu differenzieren, um dadurch eine visuelle Gruppierung der Funktionen zu ermöglichen.

Außerdem ist durch die Simulation des Programmablaufs mit dem Prototyp weiterhin aufgefallen, dass beim Navigationsmenü ein wichtiger Teil fehlte. Durch das zusätzliche Listenelement *Connect*, gelangen die Benutzer/innen wieder zurück auf den Startscreen. Ohne dieses Element müssten sie den Hardwarebutton des Gerätes verwenden und sich eventuell durch mehrere Ansichten navigieren, die vorher besucht worden waren.

2.4 Ausstellerliste

Abb. 5 Ausstellerliste

Die Ausstellerliste bietet eine Übersicht über alle Austeller der Messe. Im Prototyp wurde diese Liste ein wenig anders geplant, als sie schlussendlich umgesetzt worden ist. Für jeden Aussteller gibt es ein Listenelement mit dem Unternehmenslogo, den Namen des Unternehmens und einen Favoritenbutton, gekennzeichnet mit einem Stern.

Ursprünglich war es geplant, durch Auswahl eines Listenelements, eine kurze Beschreibung des Ausstellers einzublenden. Die Beschreibung findet sich jedoch auch im Ausstellerprofil wieder, weshalb auf die redundante Informationsdarstellung verzichtet wurde.

Nachdem ein Aussteller ausgewählt worden ist, sollte im unteren Bereich eine Buttonleiste angezeigt werden, durch die man auf die Position des Ausstellers auf dem Lageplan gelangt, das Profil angezeigt bekommt oder direkt auf die Unternehmenshomepage weitergeleitet wird. Auf die Button-leiste wurde jedoch verzichtet und die Funktionen wurden in das Ausstellerprofil übernommen.

Somit ist eine übersichtliche Liste entstanden, die zusätzlich mit zwei Buttons versehen worden ist. Über den Button mit den Standnummern gelangt man zur Position des Ausstellers am Lageplan. Durch das Aktivieren des Stern-Buttons wird der Aussteller als Favorit markiert und in der Favoritenliste angezeigt.

2.7 Messespecials

Abb. 6 Messespecials

Die Anzeige der Messespecials wurde ebenfalls als Liste geplant und realisiert. In jedem Listenelement wird ein kurzer Informationstext des Specials, sowie ein Button mit der Standnummer angezeigt, der die ursprünglich geplante Funktion in der Buttonleiste ersetzt. Somit wurde auch hier die Ansicht übersichtlicher gestaltet und die Benutzer/innen gelangen, durch den in der Liste befindlichen Button, direkt zum Lageplan. Die Standnummern wurden von der linken Seite auf die rechte Seite verschoben, um die Konsistenz mit der Ausstellerliste und den Impulsvorträgen zu gewährleisten.

2.8 Impulsvorträge

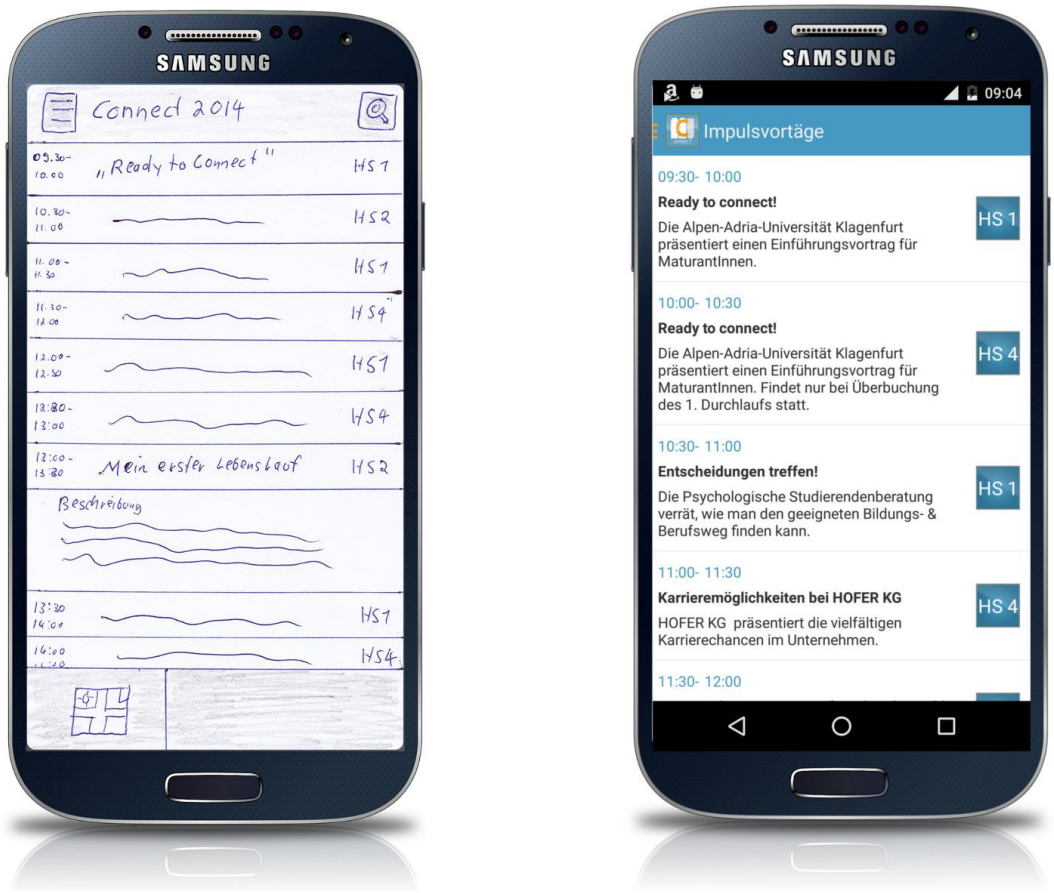

Abb. 7 Impulsvorträge

Ähnlich zu den Messespecials wurde auch die Impulsvortragsliste realisiert. Zusätzlich zu der Beschreibung und des Lageplan-Buttons, mussten auch die Überschrift und der Zeitpunkt der Vorträge in dem Listenlayout einen Platz finden. Die geplante Buttonleiste am unteren Rand wurde ebenfalls verworfen und durch Buttons in der Liste ersetzt. Die Uhrzeit und Überschrift wurde ähnlich wie im Prototyp realisiert und die Beschreibung direkt in den Listenelementen angezeigt. Die Benutzer/innen erhalten somit sofort eine Übersicht über alle Impulsvorträge und müssen nicht erst das Listenelement aktivieren, um die Beschreibung anzuzeigen.

2.9 Austellerprofil

Abb. 8 Ausstellerprofil

Das Profil der Aussteller soll wichtige Details und Informationen abbilden. Jedes Ausstellerprofil beginnt mit dem Unternehmenslogo, Favoriten- und Lageplan-Button. Zudem gibt es bei den meisten Ausstellern auch eine Kontaktperson, die bei der Messe vor Ort oder via E-Mail und Telefon erreichbar ist. Wenn verfügbar, wir ein Foto dieser Kontaktperson oder ein Platzhalterbild mit der Information, dass kein Foto vorhanden ist, angezeigt.

Neben dem Bild der Kontaktperson stehen Name, E-Mail-Adresse und Telefonnummer. Durch das Drücken auf die Telefonnummer oder E-Mail-Adresse, können die Benutzer/innen die Kontaktperson direkt aus der Applikation anrufen und eine E-Mail verschicken. Danach wird ein Link zur Homepage des Ausstellers angezeigt.

Bei allen Ausstellern ist eine Beschreibung des Unternehmens vorhanden und alle weiteren Informationen sind optional. Zum Beispiel handelt es sich bei der Alpen-Adria Universität Klagenfurt um eine Bildungseinrichtung, bei der Informationen zu Zulassungsvoraussetzungen und möglichen Abschlüssen angezeigt werden. Hingegen werden bei dem Unternehmen *Anexia* unter anderem die Branche, geplante Einstellungen und die Einsatzbereiche der Mitarbeiter angezeigt.

2.10 Lageplan

Abb. 9 Lageplan

Die Darstellung und Funktionalität des Lageplans erwies sich als einer der umfangreichsten Punkte in der Implementierung der Applikation. Am Lageplan werden die Standplätze inklusive der Standnummern angezeigt und diese sind direkt mit den Unternehmensprofilen verknüpft. Die Verknüpfung ist bidirektional, was bedeutet, dass man sowohl vom Lageplan zu den Ausstellerprofilen, als auch von den Profilen zum Lageplan gelangt. Durch das Drücken auf eine Standnummer wird den Benutzer/innen das Logo und der Name des Unternehmens angezeigt. Um zu dem Unternehmensprofil zu wechseln, muss das Logo des Ausstellers berührt werden.

Im Prototyp wurde die Funktion wiederum in einer Buttonleiste am unteren Rand geplant, jedoch in der Applikation platzsparender und intuitiver implementiert. Gelangt man vom Profil eines Ausstellers oder von einem Listenelement auf den Lageplan, so wird der Stand des Unternehmens farblich hervorgehoben.

2.11 Favoriten

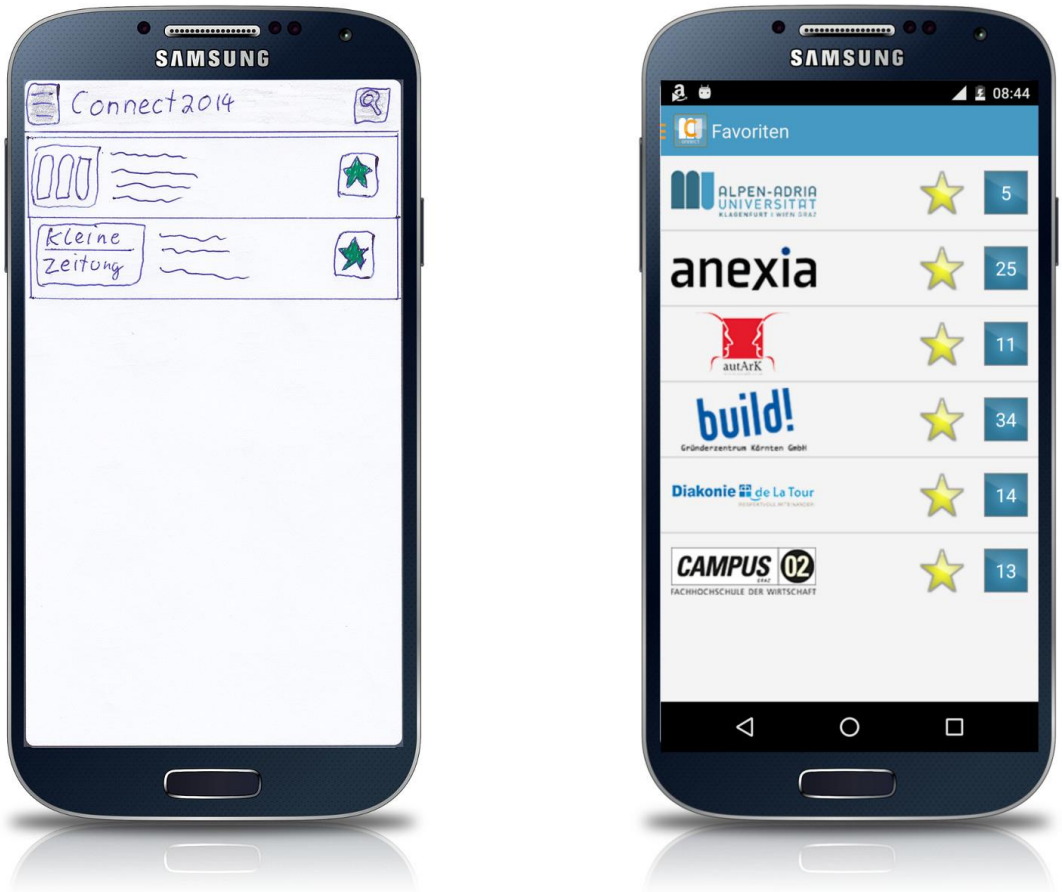

Abb. 10 Favoriten

Die Favoritenfunktion ermöglicht den Benutzer/innen, durch vorheriges Markieren der Aussteller, eine benutzerdefinierte Liste zu erstellen, um Unternehmen von besonderem Interesse schneller zu finden. Technisch gesehen handelt es sich hierbei um dieselbe Liste, die auch die gesamten Aussteller darstellt. Zusätzlich wird beim Deaktivieren des Stern-Buttons ein Bestätigungsdialog angezeigt. Nach der Deaktivierung wird der Eintrag aus der Liste gelöscht.

2.12 Zusammenfassung des Prototypingprozesses

Durch den Prototypingprozess konnten im Vorhinein viele Probleme und Designfehler erkannt und bereits vor der Implementierung beseitigt werden. Somit erhöhte sich die Usability[4] der Applikation bereits vor der Implementierungsphase und die Anzahl der Korrekturschleifen konnte erheblich reduziert werden. Obwohl die Erstellung des Prototyps, einiges an Zeit in Anspruch genommen hat, wurde durch die daraus resultierenden Ergebnisse, mindestens genau so viel Zeit in der Entwicklung wieder eingespart. Darüber hinaus war während der gesamten Entwicklungszeit, ein Modell der fertigen Applikation vorhanden, welches als Hilfestellung und Dokumentierung der erzielten Erkenntnisse diente.

3. Android

Im folgenden Abschnitt werden Mechanismen und Strukturen der Android-Programmierung, welche auch in der Connect-App verwendet werden, vorgestellt und grundlegende Begriffe wie beispielsweise *Activities*, *Fragments*, Layouts und der *Android-Lifecycle* erläutert. Außerdem wird auf die verschiedenen Möglichkeiten der Persistierung von Nutzdaten eingegangen.

3.1 MVC - Model View Controller

Die grundlegende Architektur einer Android-Applikation ist nach dem MVC-Pattern ausgelegt. Dabei handelt es sich um ein Entwurfsmuster aus der Softwareentwicklung, das zur Strukturierung von Software in drei Einheiten dient: Datenmodell (*Model*), Präsentation (*View*) und Programmsteuerung (*Controller*). Diese Architekturform ermöglicht es, einzelne Komponenten zu entwerfen, die zu einem späteren Zeitpunkt leicht wiederverwendet und erweitert werden können.

3.1.1 Model

Das *Model* repräsentiert die Geschäftsdaten und bildet die Grundlage dessen, was die Applikation verarbeitet. Es besitz keine Abhängigkeiten zu den Controllern und der Präsentationschicht und kann somit unabhängig von den beiden Komponenten entwickelt werden. Das Datenmodell der Connect-App besteht aus Ausstellern, Messespecials, Impulsvorträgen und Messeständen, mit ihren zugehörigen Attributen. Als Beispiel ist in Abb. 11 das Model eines Ausstellers zu sehen.

```
public class Aussteller{
	private int id;
	private String type;
	private String name;
	private String beschreibung;
	private String adresse;
	private String telefon;
	private String internet;
	private String email;
	private String stand_nummer;
	private String logo_name;
	private Integer is_favorite;
	private Stand stand;

	//.. Getter- und Settermethoden
}
```

Abb. 11 Datenmodell eines Ausstellers

3.1.2 View

Die *View* definiert das Design und die Präsentation der Applikation. In dieser Ebene legt man fest, in welcher Form der Inhalt der Applikation angezeigt werden soll. In der Android-Programmierung wird die View mittels *XML*[5] (*Extensible Markup Language*) definiert. Jeder Ansicht der Applikation liegt eine *XML*-Datei zugrunde, in der das Design der Ansicht definiert ist. Als exemplarisches Beispiel wird in Abb. 12, die *View* für einen Eintrag in der Liste der Impulsvorträge, in gekürzter Form dargestellt.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <TextView
        android:textColor="@color/lightblue" android:id="@id/von"
        android:text="von"
        android:layout_alignParentTop="true"
        android:layout_alignParentLeft="true"/>

    <TextView
        android:textColor="@color/lightblue"
        android:gravity="left"
        android:id="@id/bis"
        android:text="bis"
        android:layout_toRightOf="@+id/von"/>

    <Button android:textColor="@color/white"
        android:id="@id/buttonImpulsvortraegeMap"
        android:background="@drawable/map"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>

    <TextView
        android:textStyle="bold"
        android:id="@id/header"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Name des Impulsvortrages"
        android:layout_below="@+id/von"/>

    <TextView
        android:text="Beschreibung des Impulsvortrages"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/beschreibung"
        android:layout_alignParentRight="true" />
</RelativeLayout>
```

Abb. 12 XML-Layout für einen Eintrag in der Liste der Impulsvorträge

Innerhalb dieses Layouts werden die Elemente zur Anzeige eines Impulsvortrages definiert. `<TextView>`-Elemente werden dazu verwendet, statischen und dynamischen Text anzuzeigen. Durch das `<Button>`-Element wird ein Button in der Liste erstellt. Jedes Element besitzt eine eindeutige ID und mehrere Attribute, die das Erscheinungsbild, wie etwa Farbe, Größe, Schriftart etc. definieren.

Die Definition des Layouts kann jedoch nicht nur über das Erstellen von XML-Dateien realisiert werden. Müssen Daten beispielsweise asynchron geladen werden, so ist es meist notwendig, das Layout während der Laufzeit in den *Activities* zu manipulieren.

Eine weitere Methode um die *View* zu definieren, ist die Verwendung eines grafischen Entwicklungswerkzeuges. Abhängig von der verwendeten Entwicklungsumgebung werden meist auch *WYSIWYG*[7] (*What You See Is What You Get*)-Editoren angeboten, die das Erstellen und Verwalten von Layouts erheblich erleichtern. In Abb. 13 wird das Layout aus Abb. 12 mit dem Layout-Editor der Entwicklungsumgebung *Android Studio* dargestellt.

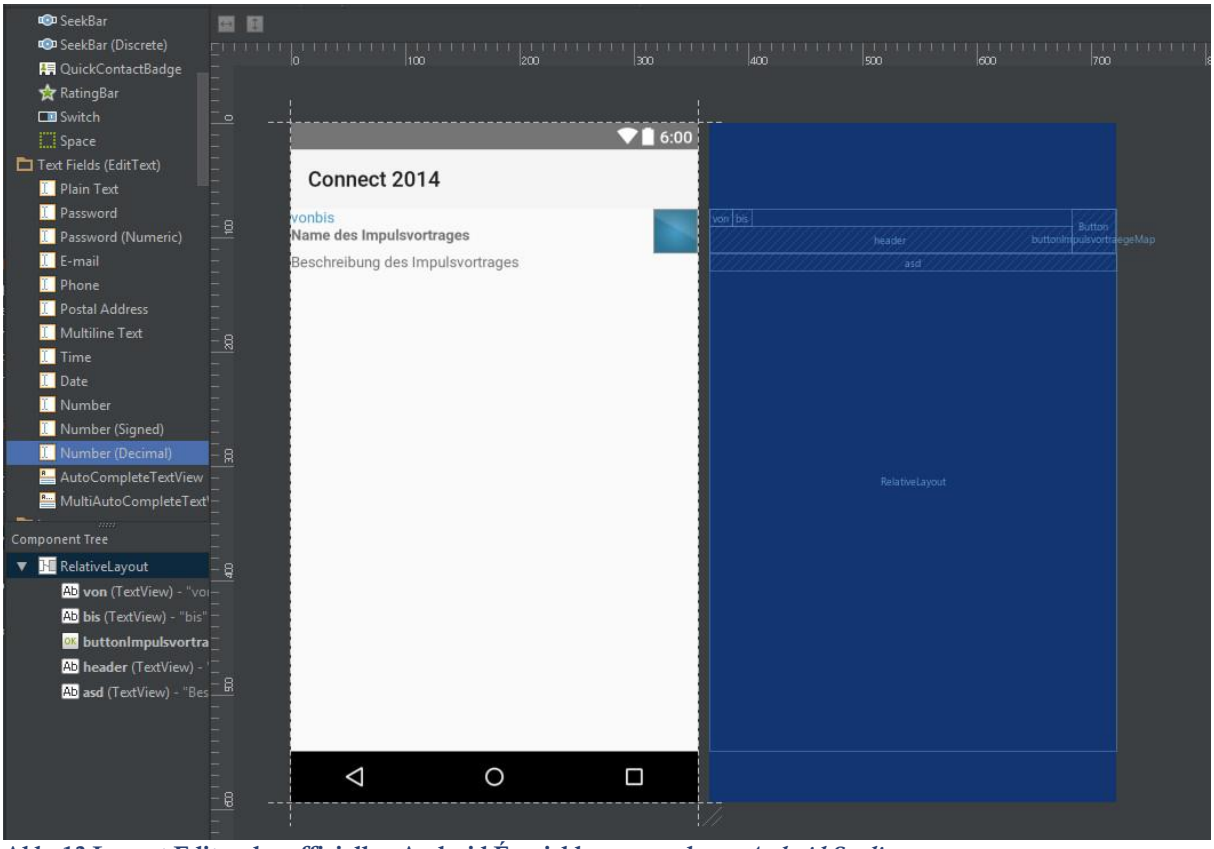

Abb. 13 Layout Editor der offiziellen Android Énwicklungsumgebung *Android Studio*

Die Definition des Layouts stellt jedoch nur eine Schablone oder Blaupause dar. Statische Elemente wie Überschriften können direkt im Layout definiert werden. Die Informationen des Datenmodells sind meist dynamisch. Um die Informationen des Datenmodells mit der Präsentationsschicht zu verknüpfen, wird eine weitere Komponente benötigt. Diese Aufgabe wird im *Controller* realisiert.

3.1.3 Controller

Ein *Controller* ist die Schnittstelle zwischen dem Datenmodell und der Präsentationsschicht. Interaktion, wie beispielsweise das Drücken eines Buttons oder die Transition von einer Ansicht zur Nächsten wird im *Controller* implementiert. Möchte ein/e Benutzer/in zum Beispiel, über den Button in der Liste der Impulsvorträge, zu dem dazugehörigen Stand des Lageplans wechseln, so wird diese Aktion im *Controller* durch eine Funktion realisiert. Die Aufgabe des *Controllers* besteht nun darin, über die Schicht des Datenmodells den richtigen Stand zu laden und an eine Instanz der Präsentationsschicht zu übergeben.

Controller werden in Android über *Activities* und *Fragments* realisiert. *Activities* und *Fragments* werden in den Abschnitten 3.2 und 3.3 näher erläutert. Die folgenden Ausschnitte der Controllerimplementierungen sollen das ganzheitliche Konzept des *MVC*-Patterns nochmals verdeutlichen und zusammenfassen.

```
public class ImpulsvortraegeFragment extends Fragment {
	// Attribute und weitere Methoden
	// ...

	@Override
	public View onCreateView(LayoutInflater i, ViewGroup g, Bundle b){
		rootView = i.inflate(R.layout.impulsvortraege, g, false);

		impDAO = getDao();
		impDAO.open();
		impulsvorträge = impDAO.getAll();
		impDAO.close();

		list = rootView.findViewById(R.id.ImpulsvortraegeList);

		ImpListAdapter adapter = new ImpListAdapter(
			rootView.getContext(), imp, this);
		list.setAdapter(adapter);

		return rootView;
	}

	// Weitere Methoden
	// ...
}
```

Abb. 14 Auszug aus dem Controller der Impulsvortragliste

In Abb. 14 ist ein Ausschnitt aus dem *Controller* der Liste der Impulsvorträge zu sehen. Die Methode `onCreateView()` wird aufgerufen, sobald der/die Benutzer/in auf die Ansicht der Impulsvorträge wechselt. In dieser Methode muss der *Controller* die benötigte View erzeugen. Das geschieht über den Aufruf `i.inflate()`. Hierbei wird unter anderem die ID des definierten Layouts übergeben, sodass eine Instanz der *View* erzeugt werden kann.

Der nächste Schritt in der Erzeugung der *View* besteht darin, die Daten aus dem Datenmodell zu laden. Das geschieht hier über eine weitere Abstraktionsebene, die sogenannte *DAO*(*Data Access Object*)- Schicht, welche den Zugriff auf Datenbankebene abstrahiert. Weiter Erläuterungen zur *DAO*-Schicht können in Abschnitt 4.2 gefunden werden.

Über den Aufruf `impDAO.getAll()` werden aus dem Datenmodell alle vorhandenen Impulsvorträge geladen. Der nächste Schritt ist die Initialisierung der Liste mit den vorhandenen Impulsvorträgen.

Über das zuvor geladene Layout wird zuerst die *View* der Liste mit dem Aufruf `rootView.findViewById()` referenziert. Danach wird ein *Adapter* erzeugt, dem die Daten der Impulsvorträge übergeben werden. Der *Adapter* ist unter anderem dafür zuständig, dass die Einträge in der Liste korrekt angezeigt werden und dass auf Interaktion seitens der Benutzer/innen reagiert wird. Die benötigte *View* besitzt nun alle Informationen und dient als Rückgabewert der `onCreateView()` Methode.

```
public class ImpListAdapter {
	// Attribute und weitere Methoden
	// ..

	@Override
	public int getCount() {
		return imp.size();
	}

	@Override
	public View getView(int pos, View convertView, ViewGroup parent) {
		View rowView = inflater.inflate(R.layout.imps_row_layout,
		parent, false);

		Button button = rowView.findViewById(R.id.button);
		TextView von = rowView.findViewById(R.id.von);
		TextView bis = rowView.findViewById(R.id.bis);
		TextView header = rowView.findViewById(R.id.header);
		TextView description = rowView.findViewById(R.id.desc);

		von.setText(imp.get(position).getVon() + "-");
		bis.setText(imp.get(position).getBis());
		header.setText(imp.get(position).getName());
		description.setText(imp.get(position).getBeschreibung());
		button.setText("HS " + Integer.valueOf(
			imp.get(position).getStandNr()));

		button.setOnClickListener(new MapScrollToPositionListener(
			this.fragment,getItemId(position),
			fragment.getActivity()));

		return rowView;
	}
	// Weitere Methoden
	// ..
}
```

Abb. 15 Auszug aus dem Adapter der Impulsvortragliste

Innerhalb des *Adapters* wird für jeden Eintrag in der Liste die richtige Information bereitgestellt und das Verhalten des Buttons für jeden Listeneintrag registriert. Die Anzahl der anzuzeigenden Elemente innerhalb der Liste wird in der Methode `getCount()` zurückgegeben. Der Adapter besitzt eine Methode `getView()`, die als Argument die Listenposition beinhaltet. Es wird nun das Layout inklusive der benötigten Elemente geladen und mit den Daten der Impulsvorträge befüllt. Mit dem Methodenaufruf `button.setOnClickListener()` wird eine weitere Hilfsklasse registriert, deren Implementierung in Kombination mit der übergebenen Position, den Übergang zum Lageplan realisiert.

3.2 Activities

Im Gegensatz zu vielen anderen Programmierparadigmen, bei denen eine Applikation in einer ausgewiesenen `main()`-Methoden gestartet wird, werden Android-Applikationen mit *Callback-Methoden* innerhalb ihrer *Activities* gestartet und gesteuert. Jede Android-Applikation muss daher mindestens eine *Activity* besitzen. Eine der *Activities* der Applikation muss als *MainActivity* ausgewiesen sein. Diese *Activity* repräsentiert den Eintrittspunkt und wird aufgerufen, wenn die Applikation gestartet wird. Alle *Activities* müssen in einer zentralen Konfigurationsdatei deklariert werden (`AndroidManifest.xml`).

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
	xmlns:android="http://schemas.android.com/apk/res/android"
	android:versionCode="1"
	android:versionName="1.0"
	package="edu.aau.connectapp">

	<uses-permission android:name="android.permission.INTERNET"/>

	<application
		android:theme="@style/AppTheme"
		android:label="@string/app_name"
		android:icon="@drawable/ic_launcher"
		android:allowBackup="true">
		<activity
			android:label="@string/app_name"
			android:name="edu.aau.connectapp.MainActivity"
			android:screenOrientation="portrait">
			<intent-filter>
				<action android:name="android.intent.action.MAIN"/>
				<category android:name="android.intent.category.LAUNCHER"/>
			</intent-filter>
		</activity>
	</application>
</manifest>
```

Abb. 16 AndroidManifest.xml der Connect-App

Innerhalb des `<application></application>`-Elements, müssen alle Activities die in der Applikation verwendet werden, mit einem `<acitivity></activity>`-Element deklariert werden. Die Connect-App besitzt lediglich eine *Activity,* die durch die Klasse `MainActivity` implementiert wird. Der Grund hierfür ist, dass das Navigationsmenü durch einen *NavigationDrawer* realisiert ist, und dieser über *Fragments* implementiert wird. Weitere Informationen hierzu werden in Abschnitt 4 näher erläutert. Durch ein `<intent-filter/>`-Element wird diese *Activity* als *MainActivity* deklariert und wird somit beim Starten der Applikation aufgerufen.

3.3 Fragments

Fragments wurden mit der Version *Android 3.0 (API Level 11)* eingeführt. Sie dienten ursprünglich dem Zweck, flexiblere Layouts auf Geräten mit größerem Bildschirm wie beispielsweise Tablets zu realisieren. In der modernen Androidentwicklung werden sie jedoch auch bei Smartphones mit kleineren Bildschirmen eingesetzt.

Unter einem Fragment kann man sich einen Abschnitt innerhalb einer *Activity* vorstellen. Dieser Abschnitt besitzt einen eigenen *Lifecycle*, verarbeitet Interaktion der Benutzer/innen und kann geändert oder gelöscht werden, während die dazugehörige *Activity* aktiv ist. Ein *Fragment* muss immer innerhalb einer *Activity* eingebettet sein und der *Lifecycle* des *Fragments* ist direkt abhängig vom dem der *Activity*. Wird eine *Activity* beispielsweise pausiert, so werden alle zugehörigen *Fragments* pausiert. Wird eine *Activity* zerstört, dann werden auch alle *Fragments* der *Activity* zerstört.

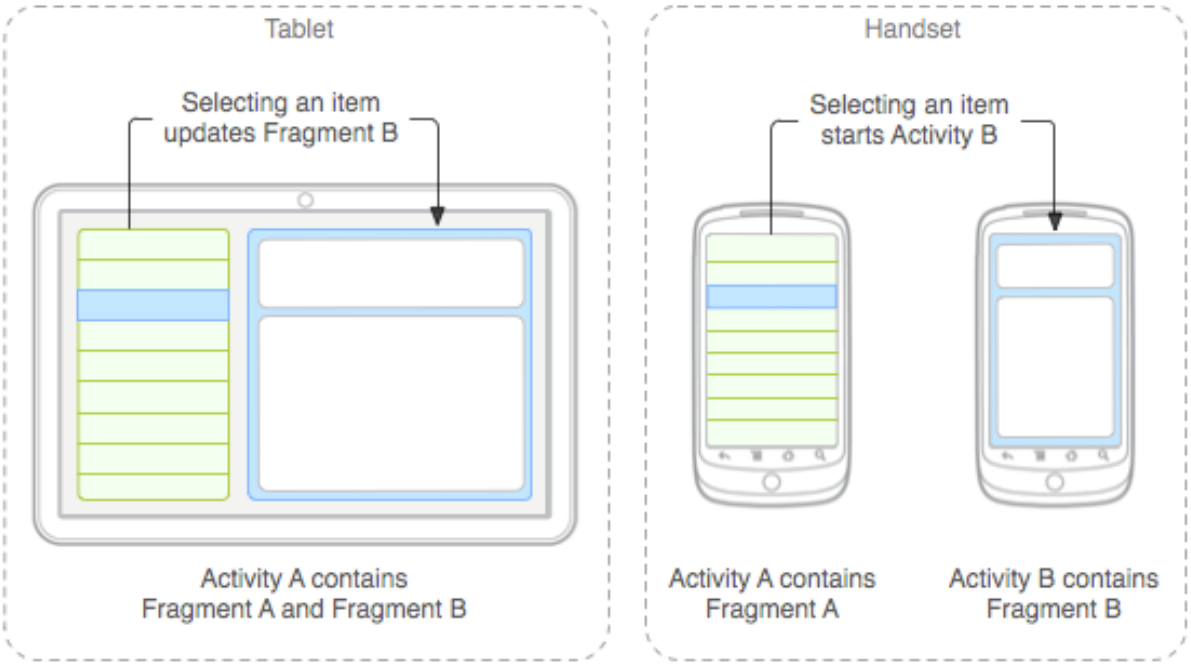

Abb. 17 Android-Fragments[6]

Abb. 17 zeigt einen Anwendungsfall für den Einsatz von *Fragments*. Für das Layout des Tablets kann innerhalb der *Activity A* das *Fragment A* und das *Fragment B* angezeigt werden. *Fragment A* dient als Navigation und *Fragment B* zur Ansicht des Inhalts. Wird nun in *Fragment* A ein anderer Navigationspunkt ausgewählt, so wird *Fragment* B durch ein *Fragment* mit dem neuen Inhalt ausgetauscht. Das gleiche Layout kann für Geräte mit kleinerem Bildschirm mit getrennten *Activities* für *Fragment A* und *Fragment B* realisiert werden.

3.4 Android-Lifecycle

Während die Benutzer/innen die Applikation starten, zwischen verschiedenen Ansichten wechseln und neue Anwendungen öffnen, durchlaufen die *Activities* und *Fragments* mehrere Statusübergänge. Wird die Applikation zum ersten Mal gestartet, erscheint diese im Vordergrund und erhält den Fokus für Interaktionen seitens der Benutzer/innen. Während dieses Vorganges ruft das Android-System mehrere *Lifecycle*-Methoden auf, in denen unter anderem das *UI* vorbereitet wird.

Eine dieser Lifecycle-Methoden wurde bereits in Abb. 14 vorgestellt. In der Methode `onCreateView()`, die von jeder *Activity* und jedem *Fragment* implementiert werden muss, wird das benötigte Layout geladen und mit den entsprechenden Daten befüllt. Diese Methode wird nur beim Erzeugen der *View* aufgerufen. Wird beispielsweise ein *Fragment* durch ein anderes ersetzt, so kann es durchaus für eine spätere Verwendung im Speicher verbleiben. Ist das der Fall und das ursprüngliche *Fragment* wird wieder geladen, so wird die Methode `onCreateView()` vom System nicht mehr aufgerufen.

Wenn während der Verwendung der Applikation, Aktionen ausgelöst werden die neue *Activities* aufrufen, so versetzt das Android-System alle anderen Activities in einen Hintergrundstatus. Diese *Activities* sind dann für die Benutzer/innen nicht mehr sichtbar. Der Zustand und die eigentliche Instanz bleiben jedoch erhalten.

Folgendes Beispiel erläutert die Transitionen zwischen Vordergrund- und Hintergrund-*Activities*:

Eine Applikation wird gestartet und die Benutzer/innen müssen sich für die Verwendung registrieren und mehrere Eingabefelder befüllen. In der Zwischenzeit wird vom Android-System ein eingehender Anruf registriert. Nun wird zuerst die Applikation pausiert, in den Hintergrund verschoben und ist nicht mehr sichtbar. Danach wird sofort die Anruf-Applikation gestartet und in den Vordergrund gebracht.

Die Benutzer/innen nehmen das Telefonat entgegen und nach dem Gespräch wird die Anruf-Applikation beendet. Das Android-System kann nun die zuletzt aktive *Activity* wieder in den Vordergrund bringen und es kann mit dem Registriervorgang, ohne Verlust der bisher eingegebenen Daten, weitergemacht werden.

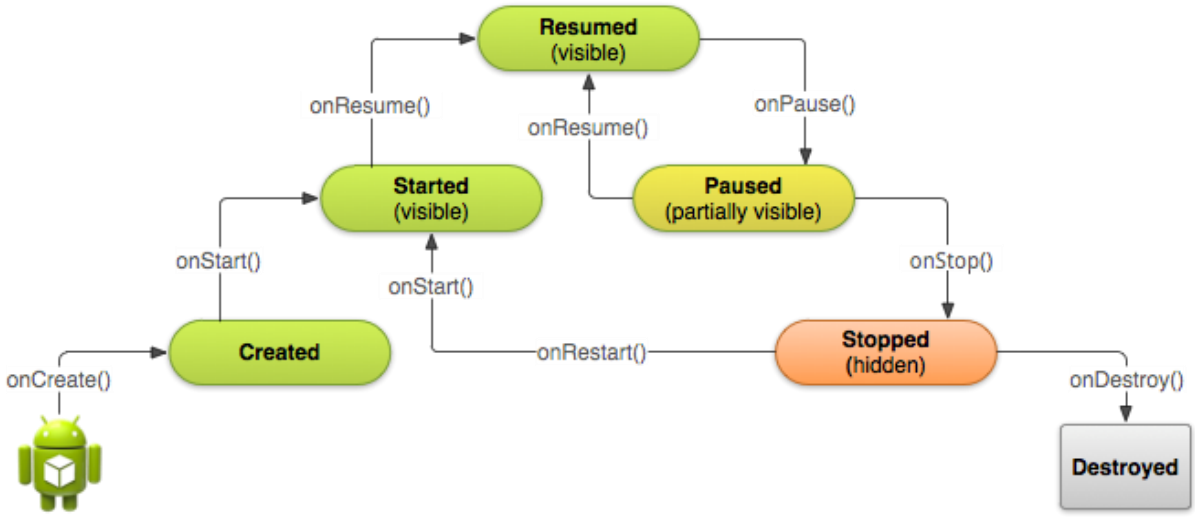

Abb. 18 Android Basic-Lifecycle[7]

Abb. 18 ist eine vereinfachte Abbildung des Android-*Lifecycle*. Während der Entwicklung von Android-Applikation müssen sich Entwickler/innen stets über die Auswirkungen von Statusübergängen bewusst sein und die Applikationen entsprechend reagieren lassen.

Ausgehend von der Komplexität der Applikation, gewährleistet eine korrekte Implementierung der Lifecycle-Methoden, dass

- die Applikation nicht abstürzt, wenn beispielsweise ein Anruf entgegengenommen werden muss oder eine andere Applikation aufgerufen wird
- während die Applikation im Hintergrund läuft, keine unnötigen Ressourcen verbraucht werden
- der Zustand der Applikation und ein eventuell vorhandener Fortschritt nicht verloren gehen
- die Applikation nicht abstürzt wenn die Ausrichtung der Anzeige von Horizontal auf Vertikal wechselt

Die Zustände des *Android-Lifecycle* werden in zwei Gruppen unterteilt, transiente und statische Zustände. Transiente Zustände können von der Applikation nicht über einen beliebigen Zeitraum beibehalten werden. Zu den transienten Zuständen zählen *Created* und *Started*. In den statischen Zuständen *Resumed*, *Paused* und *Stopped* kann die Applikation über einen längeren Zeitraum verweilen.

3.4.1 Created

Wenn die Benutzer/innen eine Applikation zum ersten Mal starten, wird vom Android-System die `onCreate()`-Methode der deklarierten *MainActivity* aufgerufen. Diese *Activity* repräsentiert somit den Startpunkt und ist mitverantwortlich für alle weiteren *Activities* und *Fragments* die aufgerufen werden. Nachdem die `onCreate()`-Methode ausgeführt wurde, befindet sich die *Activity* im Zustand *Created,* ist jedoch für die Benutzer/innen noch nicht sichtbar.

3.4.2 Started

Befindet sich eine *Activity* im Zustand *Created,* so wird sofort die `onStart()`-Methode aufgerufen. In dieser Methode können erweiterte Verhaltensweisen implementiert werden, die zum Tragen kommen, wenn die Applikation vom Status *Stopped* wieder in den Vordergrund wechselt. Die Activity wird nach dem Beenden der `onStart()`-Methode für die Benutzer/innen sichtbar und befindet sich im Zustand *Started*.

3.4.3 Resumed

Befindet sich eine *Activity* im Zustand *Started,* so wird sofort die `onResume()`-Methode aufgerufen. In dieser Methode wird das grundsätzliche Verhalten für den Übergang in den Vordergrund implementiert. Nach dem Methodenaufruf befindet sich die *Activity* im Zustand *Resumed*. Die *Activity* ist im Vordergrund und reagiert auf Aktionen der Benutzer/innen. *Resumed* ist ein statischer Zustand und kann somit über einen beliebigen Zeitraum beibehalten werden, solange kein Statusübergang eingeleitet wird.

3.4.4 Paused

Der Übergang in den Zustand *Paused* kommt zum Tragen, wenn eine neue *Activity* in den Vordergrund wechselt und die Alte nur teilweise verdrängt. Das kann der Fall sein, wenn die neue *Activity* die Alte semitransparent überdeckt oder nicht den gesamten Bildschirm beansprucht. Die verdrängte *Activity* empfängt in diesem Zustand keine Interaktion und kann auch keinen Programmcode ausführen. Der Zustand *Paused* ist ein statischer Zustand.

3.4.5 Stopped

In diesem Zustand ist die *Activity* vollkommen im Hintergrund und ist von den Benutzern/Benutzerinnen nicht sichtbar. Zwischen dem Statusübergang von *Resumed* nach *Stopped*, muss auch zwangsläufig der Status *Paused* eingenommen werden. Vom Status *Stopped* kann sich die *Activity* über die Methoden `onRestart()` und `onStart()` wieder in den Vordergrund bringen und über die Methode `onDestroy()` wird diese vollständig beendet und der *Lifecycle* der Applikation endet.

3.5 Storage

Neben der Option, Informationen direkt im Layout zu definieren, bietet die Android-Plattform mehrere Möglichkeiten für die Persistierung von Daten. Der folgende Abschnitt liefert einen kurzen Überblick über diese Möglichkeiten.

3.5.1 Shared-Preferences

Shared-Preferences werden dazu verwendet um primitive Datentypen als Key-Value Pairs zu speichern. Diese Werte sind nur von der Applikation verwendbar, die sie erstellt (*applikation-private*) und sie existieren weiter, auch wenn die Applikation beendet wurde. Sie werden unter anderem verwendet, um *User-Preferences*, also die Einstellungen die Benutzer/innen am Smartphone vornehmen, zu persistieren.

```
public class Calc extends Activity {
    public static final String PREFS_NAME = "MyPrefsFile";

    @Override
    protected void onCreate(Bundle state){
       super.onCreate(state);
       . . .

       // Restore preferences
       SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
       boolean silent = settings.getBoolean("silentMode", false);
       setSilent(silent);
    }

    @Override
    protected void onStop(){
      super.onStop();

      // We need an Editor object to make preference changes.
      // All objects are from android.context.Context
      SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
      SharedPreferences.Editor editor = settings.edit();
      editor.putBoolean("silentMode", mSilentMode);

      // Commit the edits!
      editor.commit();
    }
}
```

Abb. 19 Verwendung von Shared-Preferences[8]

Die Verwendung von *Shared-Preferences* erfolgt mithilfe der Klasse `SharedPreferences`. Innerhalb einer *Activity* erhält man über den Aufruf `getSharedPreferences()` und den Namen der Preference-Datei Zugriff darauf. Abb. 19 zeigt einen exemplarischen Anwendungsfall aus der offiziellen Android-Dokumentation.

3.5.2 Interner Speicher

Daten können in Form von Dateien im interner Speicher des Smartphones gesichert werden. Standardmäßig sind solche Dateien nur für die Applikation verwendbar die sie erstellt. Wird eine Applikation vom Smartphone entfernt, so werden auch die dazugehörigen Dateien im internen Speicher gelöscht. Der Schreib- und Lesezugriff auf das interne Dateisystem des Smartphones, verläuft ähnlich wie in einer nativen Java-Anwendung und ist in Abb. 20 abgebildet.

```
String FILENAME = "my_filename";
String text = "hello world!";
byte[] info = new byte[16];

//Lesender zugriff
FileInputStream fis = openFileInputStream(FILENAME);
fis.read(info);
fis.close()

//Schreibender Zugriff
FileOutputStream fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
fos.write(text.getBytes());
fos.close();
```

Abb. 20 Verwendung des internet Speichers

3.5.3 Externer Speicher

Jedes Android-Smartphone, hat zusätzlich zum internen Speicher noch die Möglichkeit, einen eventuell vorhandenen externen Speicher zu nützen. Dieser externe Speicher kann in Form von entfernbarer Hardware (zum Beispiel SD-Karten) oder interner Hardware (nicht entfernbar) vorliegen. Alle Dateien die im externen Speicher gesichert werden, sind *world-readable* und somit von allen Applikationen verwendbar. Sie können beliebig geändert, gelöscht und kopiert werden. Um den externen Speicher verwenden zu können, muss zuvor in der Datei `android-manifest.xml` eine entsprechende Berechtigung definiert werden. Um sowohl Lese- als auch Schreibzugriff in einer Applikation zu erhalten, verwendet man die Berechtigung `WRITE_EXTERNAL_STORAGE`, da ein schreibender Zugriff implizit auch einen lesenden Zugriff gewährleistet.

Generell empfiehlt es sich vor der Verwendung des externen Speichers eine Speicherstatusabfrage mit Hilfe von `getExternalStorageState()` auszuführen. Es kann nämlich durchaus der Fall sein, dass das Endgerät keinen externen Speicher besitzt oder dieser gerade an einem Computer verwendet wird. Somit würde die Verwendung des Speichers im schlimmsten Fall zum Absturz der Applikation führen.

3.5.4 SQLite-Datenbank

Um strukturierte Daten in Form eines Relationenschemas zu persistieren, bietet die Android-Plattform Datenbankunterstützung in Form von *SQLite an*. *SQLite* ist eine relationale Datenbank, mit einem im Vergleich zu *MySQL*[10] und *OracleDB*[10] geringeren Leistungsumfang. Da jedoch durch die Nutzung von *SQLite* nur ein vergleichsweise geringer Ressourcenaufwand benötigt wird, ist diese Datenbank auf mobilen Plattformen sehr performant.

Bei der Erstellung der Connect-App wurde ebenfalls eine *SQLite*-Datenbank vewendet. Nähre Informationen und Beispiele zur Verwendung von *SQLite* unter Android befinden sich im Abschnitt 4.2.

4. Connect-App

In diesem Abschnitt wird die Umsetzung der Connect-App im Detail betrachtet. Zuerst wird die Architektur der Applikation und das dazugehörige Datenmodell vorgestellt. Danach folgen Erläuterungen zu den Implementierungen der einzelnen Funktionen. Außerdem wird der Datenmigrationsprozess beschrieben, also der Ablauf von den Rohdaten, bis hin zur vollständigen *SQLite*-Datenbank. Abschließend wird der Veröffentlichungsvorgang im Android *Play Store* und das online stellen neuer Versionen behandelt.

4.1 Architektur

Die Applikation wurde unter Berücksichtigung der Android *Design-Guides*[10] mit einem *MVC*-Pattern umgesetzt, das in Abschnitt 3.1 bereits ausgiebig vorgestellt wurde. Zusätzlich zu den drei Komponenten des *MVC*-Patterns gibt es eine *DAO*-Schicht, die den Zugriff auf Datenbankebene abstrahiert.

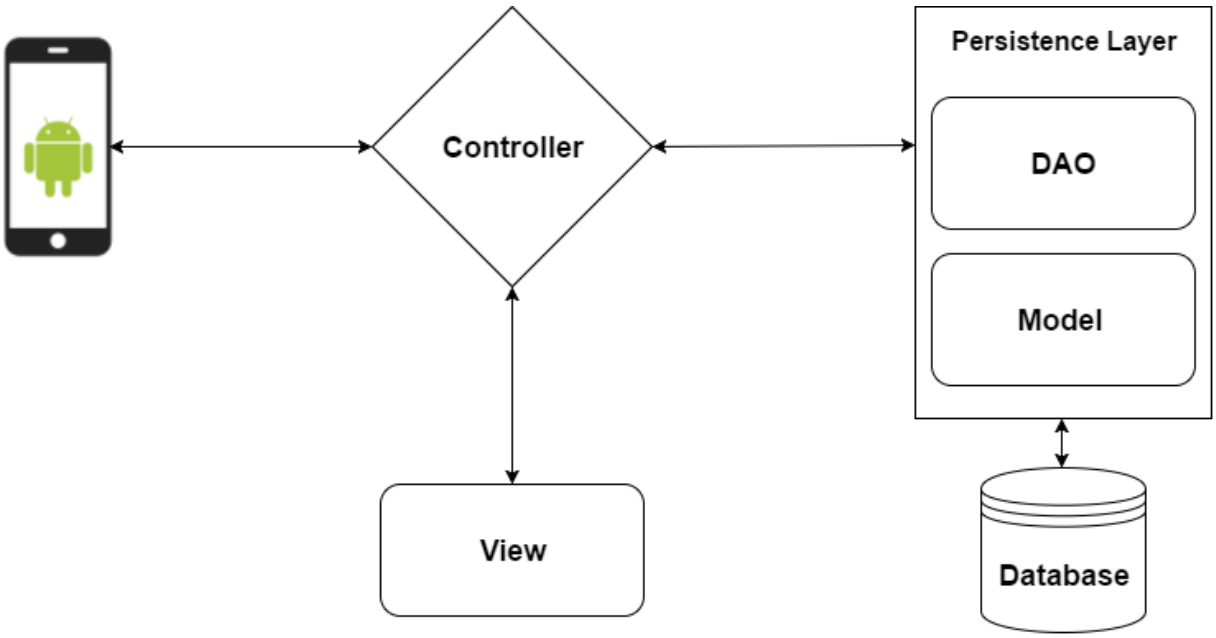

Abb. 21 Architektur der Connect-App

Die Funktionsweise, Eigenschaften und Vorteile eines MVC-Patterns bleiben mit der Einführung einer *DAO*-Schicht weiterhin erhalten und es entstehen weitere Vorteile in Hinblick auf Gliederung und Wiederverwendbarkeit. Die Kombination aus *DAO* und *Model* wird als *Persistence Layer* bezeichnet.

4.2 DAO - Data Access Model

Alle Daten der Connect-App (Aussteller, Messespecials, Impulsvorträge, etc.) werden als *SQLite*-Datenbank mit der Applikation ausgeliefert. Diese Datenbank liegt in Form der Datei `connect2014.db` im Ordner `Assets` der Applikation. Bevor diese jedoch zum ersten Mal verwendet werden kann, muss die Datenbank erst in ein Verzeichnis im Dateisystem des Smartphones kopiert werden, damit die Applikation während der Laufzeit Zugriff auf die Daten erhält. Wurde dieser Vorgang einmalig durchgeführt, so kann die App jederzeit wieder auf die Datenbank zugreifen.

Das Management der Datenbank übernimmt in der Connect-App die Klasse `MySQLiteHelper`, welche mit grundlegender Funktionalität der Android Bibliotheksklasse `SQLiteOpenHelper` ausgestattet ist und in Abb. 22 dargestellt wird.

Im Konstruktor der Klasse wird eine Referenz auf den Kontext der Applikation gesichert und es wird der Konstruktor der Superklasse `SQLiteOpenHelper`, mit dem Datenbanknamen und dem Kontext aufgerufen. In der Methode `createDatabase()` wird durch den Aufruf `checkDatabase()` überprüft, ob die Datenbank bereits in das Dateisystem kopiert wurde. Ist das der Fall, muss nichts weiter getan werden und der Zugriff auf die Datenbank kann mit `openDatabase()` erfolgen. Wird die Applikation zum ersten Mal gestartet, so ist die Datenbank noch nicht vorhanden und es wird zuerst mit `getReadableDatabase()` eine leere Datenbank im Standartverzeichnis der Applikation angelegt, die später mit der eigentlichen Datenbank überschrieben werden kann. Danach wird mit der Methode `copyDatabase()` versucht die Datenbank der Connect-App in das Dateisystem zu kopieren.

Zuerst wird über den Kontext die Datei `connect2014.db` aus dem Ordner `Assets` geladen. Danach wird die Datenbankdatei im angegebenen Verzeichnis erstellt und die Daten werden unter Zuhilfenahme von Streams von der mitgelieferten Datenbank in das Dateisystem kopiert.

Treten bei diesem Vorgang keine unvorhergesehenen Fehler auf, so ist das Endergebnis des Methodenaufrufes `createDatabase()`, immer eine funktionsfähige und vollständige Datenbank im Dateisystem. Diese Datenbank kann nun mit dem Aufruf `openDatabase()` referenziert und über das Attribut `myDataBase` angesprochen werden. Im Fall der Connect-App existiert nur eine Datenbank. Durch entsprechende Parametrisierungen von Dateinamen und Pfaden, können auch mehrere Datenbanken in einer Applikation genutzt werden.

```
public class MySQLiteHelper extends SQLiteOpenHelper {
	private static String DB_PATH = "/data/edu.aau.connectapp/databases/";
	private static String DB_NAME = "connect2014.db";
	private SQLiteDatabase myDataBase;
	private final Context myContext;

	public MySQLiteHelper(Context context) {
		super(context, DB_NAME, null, 1);
		this.myContext = context;
	}

	public void createDataBase() throws IOException{
		boolean dbExist = checkDataBase();
		if (dbExist){
			//do nothing - database already exist
		} else {
			this.getReadableDatabase();
			try {
				copyDataBase();
			} catch (IOException e) {
				throw new Error("Error copying database");
			}
		}
	}

	private boolean checkDataBase(){
		SQLiteDatabase checkDB = null;
		try{
			String myPath = DB_PATH + DB_NAME;
			checkDB = SQLiteDatabase.openDatabase(myPath, OPEN_READONLY);
		}catch(SQLiteException e){
			//database does't exist yet.
		}
		if(checkDB != null){
			checkDB.close();
		}
		return checkDB != null ? true : false;
	}

	private void copyDataBase() throws IOException{
		InputStream myInput = myContext.getAssets().open(DB_NAME);
		String outFileName = DB_PATH + DB_NAME;
		OutputStream myOutput = new FileOutputStream(outFileName);
		byte[] buffer = new byte[1024];
		int length;
		while ((length = myInput.read(buffer))>0){
			myOutput.write(buffer, 0, length);
		}
		myOutput.close();
		myInput.close();
	}

	public void openDataBase() throws SQLException{
		String myPath = DB_PATH + DB_NAME;
		myDataBase = SQLiteDatabase.openDatabase(myPath, OPEN_READONLY);
	}

	@Override
	public synchronized void close() {
		if(myDataBase != null)
			myDataBase.close();
		super.close();
	}
}
```

Abb. 22 Ausschnitt aus der Implementierung der Klasse MySQLiteHelper

Die Verwendung von Datenbanken in Android-Applikationen benötigt Ressourcen. Wenn eine Datenbank nicht permanent genutzt wird, sollte man in diesem Fall die Verbindung zur Datenbank beenden. Dies geschieht mit der Methode `close()`, welche das Schließen der Datenbankverbindung atomar implementiert.

In der Klasse `MainActivity`, also dem Einstiegspunkt der Applikation, wird jedes Mal wenn die App gestartet wird, eine neue Instanz der Klasse `MySQLiteHelper` erstellt und die Methode `createDataBase()` aufgerufen. Danach wird die Datenbankverbindung mit `openDatabase()` hergestellt und somit ist die Grundlage zur Verwendung der Datenbank vorhanden. Alle weiteren Zugriffe auf Datenbankebene erfolgen über die *DAO*-Schicht.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
	super.onCreate(savedInstanceState);
	setContentView(R.layout.activity_main);

	/**
	 * Checks if db already exists and creates the db if necessary
	 */
	sqLiteHelper = new MySQLiteHelper(this);
	try {
		sqLiteHelper.createDataBase();
	} catch (IOException e) {
		Log.e(TAG, "Database creation error");
	}
	try{
		sqLiteHelper.openDataBase();
	}catch(SQLException sqlex){
		Log.e("TAG","OpenDataBase error");
	}

	....
}
```

Abb. 23 Verwendung der Klasse MySQLiteHelper innerhalb der MainActivity

Als exemplarisches Beispiel für eine *DAO*-Implementierung der Connect-App wird in weiterer Folge die Verwendung der Klasse `AusstellerDAO` präsentiert. Für alle anderen Entitäten des Models existiert eine analoge *DAO*-Implementierung. In Abb. 24 wird der erste Ausschnitt dieser Klasse gezeigt. Jedes *DAO* besitzt eine Instanz der in Abb. 22 vorgestellte Hilfsklasse `MySQLiteHelper`, die beim Erzeugen des *DAOs* im Konstruktor erstellt wird und eine Referenz auf die eigentliche Datenbank darstellt. Die beiden Attribute `allColumnsAussteller` und `allColumnsStand` dienen als Referenz für die Spalten in den Datenbanktabellen.

Der Zugriff auf Zeilen und Spalten in der Datenbank geschieht über `Cursor`-Objekte. In der Methode `getAllAussteller()` wird zum ersten Mal ein solches `Cursor`-Objekt verwendet. Diese Methode gibt alle vorhandenen Aussteller in der Datenbank, als Objekte des Typs Aussteller zurück. Ein `Cursor`-Objekt beinhaltet alle Zeilen, die nach einer Datenbankabfrage als Ergebnis zurückgeliefert werden.

Durch den Aufruf von `query(MySQLiterHelper.TABLE_NAME_AUSSTELLER, allColumnsAussteller)` wird ein `Cursor` erstellt, der alle Zeilen, projiziert auf alle Spalten in der Tabelle Aussteller beinhaltet. Möchte man beispielsweise nur die ID und den Namen der Aussteller abfragen, so gibt man anstatt `allColumnsAussteller`, nur die Spalten für ID und Namen an.

```
public class AusstellerDao {

	private SQLiteDatabase database;
	private MySQLiteHelper dbHelper;

	private String[] allColumnsAussteller = {
			MySQLiteHelper.COLUMN_ID,
			MySQLiteHelper.COLUMN_TYPE,
			MySQLiteHelper.COLUMN_NAME,
			MySQLiteHelper.COLUMN_BESCHREIBUNG,
			....
			};

	private String[] allColumnsStand = {
			....
	};

	public MySQLiteHelper getDbHelper() {
		return dbHelper;
	}

	public AusstellerDao(Context context) {
		dbHelper = new MySQLiteHelper(context);
	}

	public void open() throws SQLException {
		database = dbHelper.getWritableDatabase();
	}

	public void close() {
		dbHelper.close();
	}

	public List<Aussteller> getAllAussteller() {
		List<Aussteller> aussteller = new ArrayList<Aussteller>();

		Cursor cursor = database.query(MySQLiteHelper.TABLE_NAME_AUSSTELLER,
		allColumnsAussteller);

		cursor.moveToFirst();
		while (!cursor.isAfterLast()) {
			Aussteller aus = cursorToAussteller(cursor);
			aussteller.add(aus);
			cursor.moveToNext();
		}

		cursor.close();
		return aussteller;
	}

	....

	private Aussteller cursorToAussteller(Cursor cursor) {
			Aussteller aus = new Aussteller();
			aus.setId(cursor.getInt(0));
			aus.setType(cursor.getString(1));
			aus.setName(cursor.getString(2));
			....
			return aus;
		}

}
```

Abb. 14 Ausschnitt aus der Implementierung der Klasse AusstellerDAO

Nachdem der `Cursor` erzeugt wurde, wird über die Tabellenzeilen mit Hilfe von `cursor.moveToNext()` iteriert und für jede dieser Zeilen ein Aussteller-Objekt erzeugt. Das Erzeugen der Aussteller-Objekte wurde in die Hilfsmethode `cursorToAussteller()` ausgelagert, da diese Funktionalität an mehreren Stellen in der Klasse verwendet wird.

Das Abrufen der Spalten, und damit den eigentlichen Attributen der Aussteller, geschieht mit den Methodenaufrufen `cursor.getInt(0)`, `cursor.getString(1)`, etc. Basierend auf der Position der Spalten in der Tabelle und den dazugehörigen Datentypen, werden die Attribute in die Aussteller geladen. Ein Aussteller-Objekt (Abb. 26) besitzt neben der Funktion `compareTo()`, welches durch das Interface `Comparable` realisiert wird, lediglich Attribute und dazugehörige *Getter*- und *Setter*-Methoden.

```
public class AusstellerDao {
	....

	public Aussteller findAusstellerByName(String searchString) {
		Aussteller result = null;

		Cursor cursor = database.query(MySQLiteHelper.TABLE_NAME_AUSSTELLER,
		allColumnsAussteller, MySQLiteHelper.COLUMN_NAME + "like" +
		searchString);

		cursor.moveToFirst();
		result = cursorToAussteller(cursor);
		cursor.close();

		return result;
	}

	public Aussteller findAusstellerByStand(String nr) {
		Aussteller result = null;
		Cursor cursor = database.query(MySQLiteHelper.TABLE_NAME_AUSSTELLER,
		allColumnsAussteller, MySQLiteHelper.COLUMN_STAND_NUMMER + "like " +
		nr);

		cursor.moveToFirst();
		if(cursor.getCount() != 0){
			result = cursorToAussteller(cursor);
		}

		cursor.close();
		return result;
	}

	....
}
```

Abb. 25 Ausschnitt aus der Implementierung der Klasse AusstellerDAO

In Abb. 25 werden zwei Methoden implementiert, die eine Datenbankabfrage realisieren. Die Methode `findAusstellerByName()` liefert einen Aussteller zurück, dessen Name dem Parameter `searchString` entspricht. Der Unterschied zur Methode `getAllAussteller()` besteht hier darin, dass zusätzlich zum Tabellennamen und den Spalten, noch eine Abfrage als dritter Parameter mitgegeben wird.

Der Aufruf von:

```
query(.TABLE_NAME_AUSSTELLER, allColumnsAussteller,
.COLUMN_NAME + "like" + searchString)
```

ist also äquivalent zu der Datenbankabfrage:

```
Select * from Aussteller where name = :searchstring.
```

Analog wird in der Methode `findAusstellerByStand()` ein Aussteller auf Basis der Standnummer geladen und zurückgegeben.

```
public class Aussteller implements Comparable<Aussteller> {
	private int id;
	private String type;
	private String name;
	private String beschreibung;
	private String adresse;
	private String telefon;
	.....

	//Getter- und Setter-Methoden

	@Override
	public int compareTo(Aussteller another) {
		return this.getName().toLowerCase().compareTo(
		another.getName().toLowerCase());
	}
}
```

Abb. 26 Ausschnitt aus der Implementierung der Klasse Aussteller

In Abb. 27 ist der eigentliche *DAO*-Aufruf aus der *Controller*-Komponente ersichtlich. In der Methode `onCreateView()` wird im `AusstellerListFragment`, welches der Controller für die Listenansicht der Aussteller ist, ein `AusstellerDAO` erzeugt. Durch den Aufruf von `getAllAussteller()` wird über die entsprechende DAO-Implementierung eine Liste von allen Ausstellerobjekten zurückgegeben. Diese Liste wird anschließend verwendet um die Aussteller anzuzeigen.

Hier wird nun der Vorteil einer *DAO*-Komponente ersichtlich. Der Controller muss lediglich die entsprechende *DAO*-Methode aufrufen, um die Daten des *Models* abzurufen und ist somit völlig unabhängig von der Logik der Datenbankimplementierung.

```
public class AusstellerListFragment extends Fragment {
	private View rootView;
	private AusstellerDao aDao;
	private List<Aussteller> aussteller;
	private ListView list;

	@Override
	public View onCreateView(LayoutInflater inflater, ViewGroup container,
	Bundle savedInstanceState) {
		rootView = inflater.inflate(R.layout.fragment_aussteller_list,
		container, false);

		aDao = getDao();
		if (getArguments().getBoolean("search")){
			aussteller = SearchListener.getResults();
		} else
			aussteller = getAllAussteller();

		//GetList and set adapter/onClickListener
		list = (ListView) rootView.findViewById(R.id.ausstellerList);

		AusstellerListAdapter adapter =
		new AusstellerListAdapter(rootView.getContext(), aussteller, this,
		false);

		list.setAdapter(adapter);
		list.setOnItemClickListener(...);

		return rootView;
	}

	/**
	 * @return all Aussteller in db sortet by Ausstellername
	 */
	private List<Aussteller> getAllAussteller(){
		ArrayList<Aussteller>aussteller = new ArrayList<>();
		aDao.open();
		aussteller = (ArrayList<Aussteller>) aDao.getAllAussteller();

		aDao.close();
		Collections.sort(aussteller);
		return aussteller;

	}

	protected AusstellerDao getDao(){
		return new AusstellerDao(rootView.getContext());
	}
}
```

Abb. 27 Datenbankzugriff durch die Klasse AusstellerDAO

4.3 Datenmodell

In Abb. 28 wird das Datenmodell der Connect-App abgebildet. Ausgehen von der Anforderungsanalyse, wurden vier Entitäten identifiziert, die in der Datenbank abgebildet werden. Der größte Teil des Modells bezieht sich auf die Aussteller, die an verschiedenen Ständen, Unternehmen und Einrichtungen repräsentieren. Es gibt drei verschiedene Typen von Ausstellern. Typ *A* stellt Unternehmen dar und wird im Modell als *Company* bezeichnet. Unternehmen besitzen spezifische Attribute, die sie von anderen Ausstellern unterscheiden.

Zu diesen Attributen zählen die Branchen, in denen das Unternehmen tätig ist, Standorte, die Anzahl der Mitarbeiter/innen, geplante Einstellungen und Einsatzbereiche. Der zweite Typ von Ausstellern (Typ *B*) stellt Bildungseinrichtungen dar. Spezielle Attribute die nur Bildungseinrichtungen besitzen sind Zulassungsvoraussetzungen, Abschlüsse und Studiengänge. Typ *A* und Typ *B* Austeller besitzen darüber hinaus auch gemeinsame Attribute wie beispielsweise Namen, Beschreibung, Adresse und E-Mail. Als Typ *C* werden Aussteller charakterisiert, die weder Bildungseinrichtungen, noch Unternehmen sind. Diese Aussteller besitzen meist nur einen Namen, Beschreibung und ein Logo.

Um ein Minimum an Konsistenz in Bezug auf die Darstellung der Aussteller zu gewährleisten, wurde vorab beschlossen, dass jeder Aussteller mindestens einen Namen, eine Beschreibung und ein Logo besitzen muss. Außerdem gibt es Aussteller, die Bilder von Ansprechpersonen in ihren Profil besitzen. Logos und Bilder von Ansprechpersonen werden jedoch nicht direkt in der Datenbank gespeichert. Diese Dateien werden ähnlich wie Icons und andere Grafiken im Verzeichnis `drawable` mit der Applikation ausgeliefert. In der Datenbank werden in den Feldern `logo_name` und `ansprechperson_bild` die Dateinamen der Bilder gespeichert.

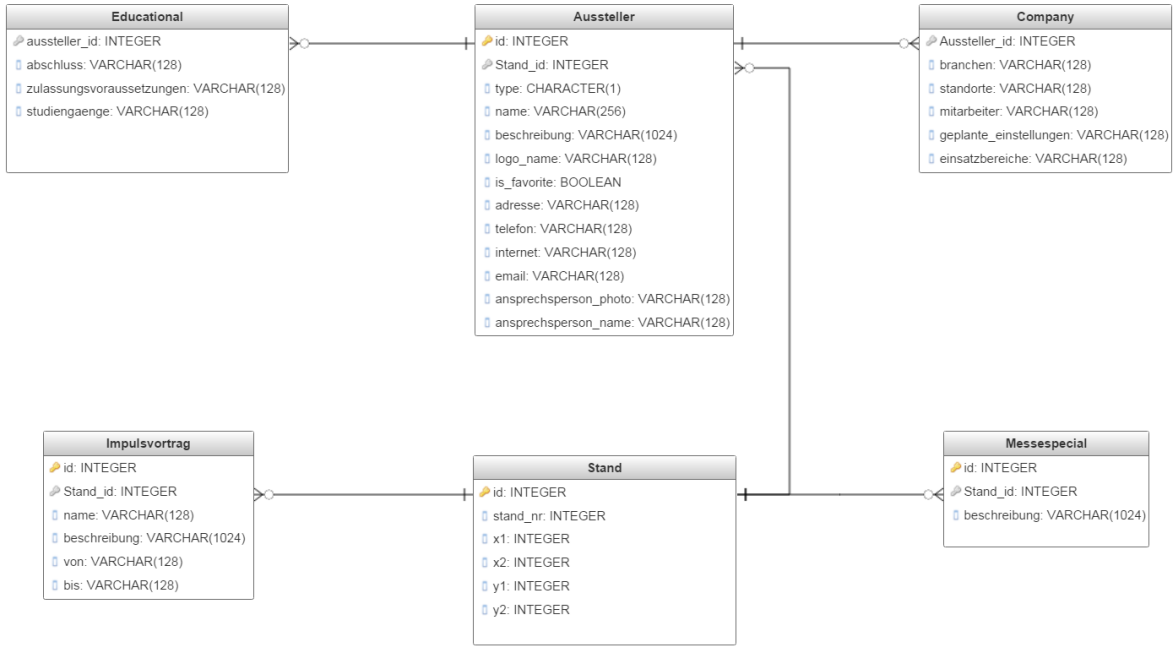

Abb. 28 Datenmodell der Connect-App

Bei der zweiten Entität handelt es sich um den Impulsvortrag. Impulsvorträge besitzen einen Namen, eine Beschreibung und einen Zeitraum indem sie abgehalten werden. Messespecials besitzen lediglich eine Beschreibung.

Als Letztes werden noch die Messestände im Datenmodell abgebildet. Jeder Aussteller, Impulsvortrag und jedes Messespecial ist einem Stand zugeordnet. Jeder Stand besitzt eine Nummer, die am Lageplan der Messe ersichtlich ist. Außerdem werden die Pixelkoordinaten der Stände am Bild des Lageplans gespeichert, um den interaktiven Lageplan der Messe-App umsetzen zu können. Nähere Informationen dazu befinden sich im Abschnitt 4.10.

4.4 App-Navigation

Das Navigationskonzept der Connect-App wird unter Android als *NavigationDrawer* bezeichnet. Durch eine Wischgeste von links nach rechts wird eine Liste angezeigt, die alle Menüpunkte der Applikation darstellt. Diese Liste ist von jedem Menüpunkt aus abrufbar und somit können die Benutzer/innen jederzeit alle Funktionen der Applikation nutzen. Alternativ kann dieses Navigationsmenü auch über den Button mit den drei horizontalen Linien in der linken oberen Ecke aufgerufen werden. Aufgrund der Konvention, den Button für dieses Menü mit drei horizontalen Linien zu kennzeichnen wird es auch häufig als *Hamburger-Menü* bezeichnet. Darstellungen des Navigationskonzeptes können in Abb.2 und Abb.4 vorgefunden werden.

In Abb. 29 befindet sich ein Ausschnitt aus der Implementierung des *NavigationDrawer*-Menüs. Da das Navigationsmenü von jeder Ansicht aus abrufbar sein soll, muss die Implementierung somit in der *MainActivity* erfolgen. Wie in Abb. 29 ersichtlich, implementiert die *MainActivity* das Interface `NavigationDrawerFragment`, welches alle Methoden die für die Umsetzung der Navigation erforderlich sind, vorgibt. In der `onCreate()`-Methode wird zuerst ein *NavigationDrawer*-Objekt und das dazugehörige Layout erstellt. Danach wird dem *NavigationDrawer* das Layout mit der Methode `setUp()` zugewiesen.

In der Methode `onNavigationDrawerItemSelected()`, welche im Interface `NavigationDrawerFragment` spezifiziert ist, wird das Verhalten der Auswahl eines Menüpunktes implementiert. Als Parameter wird dieser Methode die Position des Elementes in der Liste übergeben, mit der das benötigte Layout geladen werden kann. Durch die Klasse `FragmentManager` und den Aufruf `replace()` wird der Übergang von einer Ansicht zur Nächsten realisiert. Wird die Methode beispielsweise mit dem Parameter *2* aufgerufen, so haben die Benutzer/innen den Menüpunkt Impulsvorträge gewählt, welcher an dritter Stelle in der Liste platziert ist.

Erfolgt ein Austauschen von *Fragments* durch den `FragmentManager` so wird automatisch die Methode `onSectionAttached()` aufgerufen. In dieser Methode wird der Titel der Ansicht, welcher in der linken oberen Ecke angezeigt wird, aktualisiert. Die einzelnen Implementierungen der Ansichten werden in den folgenden Abschnitten präsentiert.

```
public class MainActivity extends Activity
implements NavigationDrawerFragment {
	...

	@Override
	protected void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
		setContentView(R.layout.activity_main);
		...
		mNavigationDrawerFragment = findFragmentById(R.id.drawer);
		mDrawerLayout = findViewById(R.id.drawer_layout);
		mNavigationDrawerFragment.setUp(
				R.id.navigation_drawer,
				mDrawerLayout);
		...
	}

	@Override
	public void onNavigationDrawerItemSelected(int p) {
		FragmentManager fragmentManager = getFragmentManager();
		switch(p){
		case 0 :	//Info Page
			fragmentManager.beginTransaction()
			.replace(R.id.container, InfoFragment.newInstance(p))
			.commit();
			break;
		case 1 :	//Aussteller List
			fragmentManager.beginTransaction()
			.replace(R.id.container,
			AusstellerListFragment.newInstance(p))
			.commit();
			break;
		case 2 :	//Impulsvortraege List
			fragmentManager.beginTransaction()
			.replace(R.id.container,
			ImpulsvortraegeFragment.newInstance(p))
			.commit();
			...
		}
	}

	public void onSectionAttached(int number) {
		switchSection(number);
	}

	private void switchSection(int number){
		switch (number) {
		case 0:
			mTitle = getString(R.string.title_info);
			break;
		case 1:
			mTitle = getString(R.string.title_aussteller);
			break;
		case 2:
			mTitle = getString(R.string.title_impulsvortraege);
			break;
		...
	}
	...
}
```

Abb. 29 Ausschnitt aus der Implementierung der Klasse NavigationDrawerFragment

4.5 Startbildschirm

Beim Startbildschirm der Applikation werden lediglich statische Informationen, welche in der Layoutdatei `fragment_main.xml` (Abb.30) festgelegt sind, angezeigt. Dazu zählen das Logo der Messe, Ort und Zeit der Veranstaltung, die Webseite und ein Abschnitt mit den Logos der größten Sponsoren der Messe.

In Abb. 31 ist ersichtlich, dass das definierte Layout in der `onCreateView()`-Methode instanziert und zurückgegeben wird. Wird nun in der zuvor besprochenen *MainActivity* auf den Startbildschirm gewechselt, so wird automatisch innerhalb der Methode `newInstance()` eine neue Instanz der Klasse erzeugt. Die Position im Navigationsmenü (`sectionNumber`) wird als Argument für die spätere Verwendung gesichert.

Innerhalb der Methode `onAttach()` wird dieses Argument mit `getArguments().getInt()` wieder geladen und durch den Aufruf der Methode `onSectionAttached` wird der Austausch der Überschrift realisiert.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.aau.connectapp.MainActivity" >

    <ImageView
        android:id="@+id/imageView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:src="@drawable/connect_logo2014" />

    <TextView
        android:id="@+id/wwwconnect"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/place"
        android:layout_below="@+id/place"
        android:layout_marginTop="20dp"
        android:text="@string/www_connect"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:textColor="@color/deepblue"
        android:autoLink="web" />

    <TextView
        android:id="@+id/place"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/time"
        android:layout_below="@+id/time"
        android:layout_marginTop="15dp"
        android:text="@string/place_connect"
        android:textAppearance="?android:attr/textAppearanceMedium"
        android:textColor="@color/deepblue" />

    <TextView
        android:id="@+id/time"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/date"
        android:layout_below="@+id/date"
        android:text="@string/time_connect"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:textColor="@color/deepblue" />

    <TextView
        android:id="@+id/date"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/imageView1"
        android:layout_below="@+id/imageView1"
        android:layout_marginTop="38dp"
        android:text="@string/date_connect"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:textColor="@color/deepblue" />

</RelativeLayout>
```

Abb. 30 Layoutdatei fragmen_main.xml

```
public class InfoFragment extends Fragment {
	public static InfoFragment newInstance(int sectionNumber) {
		InfoFragment fragment = new InfoFragment();
		Bundle args = new Bundle();
		args.putInt(MainActivity.ARG_SECTION_NUMBER, sectionNumber);
		fragment.setArguments(args);
		return fragment;
	}

	@Override
	public View onCreateView(LayoutInflater inflater, ViewGroup
		container, Bundle savedInstanceState) {
		View rootView = inflater.inflate(R.layout.fragment_main,
			container);
		return rootView;
	}

	public void onAttach(Activity activity) {
		super.onAttach(activity);
		activity.onSectionAttached(
			getArguments().getInt(
			MainActivity.ARG_SECTION_NUMBER
		);
	}
}
```

Abb. 31 Ausschnitt aus der Implementierung der Klasse InfoFragment

4.6 Aussteller

Die Implementierungen der Ausstellerliste, Ausstellerprofile und der dazugehörigen Suche sind mitunter die umfangreichsten der gesamten Applikation. Im Abschnitt 4.2 und in Abb. 27 wurde bereits ein Teil aus dem *Fragment* der Ausstellerliste präsentiert. Nun werden die listenspezifischen Methoden, Klassen und Layoutdateien vorgestellt und das Ausstellerprofil genauer betrachtet.

```
public class AusstellerListFragment extends Fragment {
      // Attribute
      ...

      @Override
      public View onCreateView(LayoutInflater inflater, ViewGroup
            container, Bundle savedInstanceState) {

            rootView = inflater.inflate(
            R.layout.fragment_aussteller_list,container);

            //Dao- Zugriff und laden der Aussteller aus der Datenbank
            ...

            list = (ListView) rootView.findViewById(R.id.ausstellerList);

            AusstellerListAdapter adapter = new AusstellerListAdapter(
            rootView.getContext(), aussteller ...);

            list.setAdapter(adapter);

            list.setOnItemClickListener(new CompanyListListener(
            this, rootView.getContext(),(MainActivity)getActivity()));

            ...

            return rootView;

      }
      ...
}
```

Abb. 32 Listenspezifische Implementierung der Klasse AusstellerListFragment

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#f4f4f4"
    android:orientation="vertical" >

    <ListView
        android:id="@+id/ausstellerList"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:dividerHeight="2dp">
    </ListView>
</RelativeLayout>
```

Abb. 33 Layoutdatei fragment_aussteller_list.xml

In Abb. 32 ist ein Teil der Klasse `AusstellerListFragment` abgebildet, welcher für die Erstellung der Ausstellerliste zuständig ist. Zuerst wird das Layout der Ausstellerliste geladen, welches lediglich eine Definition einer `<ListView/>` enthält (Abb. 33).

Danach wird diese `ListView` mit Hilfe der Methode `findViewById()` referenziert. Die nächsten beiden Schritte sind die Erstellung eines Adapters und eines *OnClickListeners* für die Ausstellerliste. Beide Objekte werden der Ausstellerliste mit `setAdatper()` und `setOnClickListener()` zugewiesen.

Der Adapter `AusstellerListAdapter` ist für die Listenlogik verantwortlich und sorgt dafür, dass jeder Eintrag in der Liste korrekt angezeigt wird und das gewünschte Verhalten aufweist.

Der *OnClickListener* `CompanyListListener()` steuert das Verhalten, wenn ein Listenelement ausgewählt wird. Implementierungen des Adapters und des *OnClickListeners* sind in Abb. 34 und Abb. 36 zu finden.

Beim Erstellen des Objekts `AusstellerListAdapter` werden der `Context`, die Ausstellerobjekte und das `AusstellerListFragment` als Referenzen gespeichert. Über den `Context` ist es dem Adapter möglich, Zugriff auf Funktionalität einer Activity zu erlangen. Dies ist notwendig, um innerhalb des Adapters einen `LayoutInflater` und mit dessen Hilfe das Layout der einzelnen Listenelemente zu laden. In den Ausstellerobjekten sind die Informationen der einzelnen Aussteller vorhanden und das `AusstellerFragment` muss an einigen Stellen an weitere *OnClickListener* übergeben werden.

In den Methoden `getItem()` und `getItemId()` werden die Aussteller und deren IDs aufgrund der Position in der Liste zurückgeliefert.

```
public class AusstellerListAdapter extends BaseAdapter {
	//Attribute
	...

	public AusstellerListAdapter(Context context, List<Aussteller> aussteller,
		AusstellerListFragment fragment){
		this.context = context;
		this.aussteller = aussteller;
		this.fragment = fragment;
	}

	@Override
	public int getCount() {
		return this.aussteller.size();
	}

	@Override
	public Object getItem(int position) {
		return aussteller.get(position);
	}

	@Override
	public long getItemId(int position) {
		return aussteller.get(position).getId();
	}

	@Override
	public View getView(int position, View convertView, ViewGroup parent) {
		...
		View rowView = inflater.inflate(R.layout.ausstellerrowlayout, ...);

		ImageView ausstellerLogo =rowView.findViewById(R.id.ausstellerLogo);
		Button ausstellerMapButton = rowView.findViewById(
		R.id.buttonAusstellerMap);

		ausstellerFavoriteButton = rowView.findViewById(
		R.id.buttonAusstellerFavorite);

		ausstellerFavoriteButton.setOnClickListener(
		new OnFavoriteClickListener(...));
		ausstellerMapButton.setText(aussteller.get(position)
		.getStand_nummer());

		ausstellerLogo.setImageResource(logoResID);
		ausstellerMapButton.setOnClickListener(
		new MapScrollToPositionListener(...));
		checkFavorite(position);
		return rowView;
	}

	private void checkFavorite(int position){
		if(aussteller.get(position).getIs_favorite() == 0){
			ausstellerFavoriteButton.setBackgroundResource(starResID);
		} else
			ausstellerFavoriteButton
			.setBackgroundResource(starFavoriteResID);
	}
}
```

Abb. 34 Ausschnitt aus der Implementierung der Klasse AusstellerListAdapter

Die Methode `getView()` wird aufgerufen, wenn die Listenelemente erzeugt werden. Zuerst wird mit Hilfe des `LayoutInflater` das Layout für einen Listeneintrag geladen (Abb. 34). Danach werden die beinhaltenden Elemente referenziert und mit den Informationen des entsprechenden Ausstellers befüllt.

Diese Informationen sind im Fall der Listenansicht, das Ausstellerlogo (`ausstellerLogo`), ein Favoriten-Button (`ausstellerFavoriteButton`) und ein Lageplan-Button (`ausstellerMapButton`). Das Ausstellerlogo wird mithilfe der Methode `setImageResource()` und der dazugehörigen ID der `ImageView` zugewiesen. Für die beiden Buttons wird in weiterer Folge ein entsprechender *OnClickListener* registriert. In der Methode `checkFavorite()` wird überprüft, ob die Aussteller bereits zu den Favoriten zählen und ausgehend davon, wird ein goldener oder grauer Stern angezeigt.

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ImageView
        android:id="@+id/ausstellerLogo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_centerVertical="true"
        android:focusable="false"
        android:maxHeight="10dp"
        android:maxWidth="100dp"
        android:paddingLeft="10dp"
        android:paddingTop="5dp"
        android:paddingBottom="5dp"
        android:src="@drawable/joham_und_partner"/>

    <Button
        android:id="@+id/buttonAusstellerMap"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_centerVertical="true"
        android:background="@drawable/map"
        android:focusable="false"
        android:gravity="center"
        android:minHeight="0dp"
        android:minWidth="0dp"
        android:textColor="@color/white"
        android:layout_marginRight="10dp"/>

    <Button
        android:id="@+id/buttonAusstellerFavorite"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerVertical="true"
        android:layout_toLeftOf="@+id/buttonAusstellerMap"
        android:background="@drawable/star"
        android:focusable="false"
        android:minHeight="0dp"
        android:minWidth="0dp"
        android:layout_marginRight="15dp"/>
</RelativeLayout>
```

Abb. 35 Layoutdatei ausstellerrowlayout.xml

Die Klasse `CompanyListListener` ist dafür zuständig, dass nach der Auswahl eines Listenelementes das richtige Ausstellerprofil angezeigt wird. Im Abschnitt 4.3 wurde erwähnt, dass es drei unterschiedliche Typen von Ausstellern gibt. Bei der Auswahl eines Listenelementes wird in der `onItemClick()`-Methode der Aussteller geladen und es wird überprüft, zu welchem Typ dieser Aussteller gehört.

Da alle drei Typen von Ausstellern teilweise unterschiedliche Informationen besitzen, existieren für jeden Ausstellertyp auch ein eigenes Layout und ein eigenes *ProfileFragment*. Ist der Aussteller beispielsweise ein Unternehmen, wird durch den `FragmentManager` das aktuelle *Fragment,* durch das *Fragment* `CompanyProfileFragment` ersetzt, welches in weiterer Folge das Ausstellerprofil mithilfe der ID des Ausstellers ladet. Die Fragments der einzelnen Ausstellertypen registrieren wiederum *OnClickListener* für Favoriten-Buttons, Lageplan-Buttons etc.

```
public class CompanyListListener implements OnItemClickListener{
	//Attribute
	...

	@Override
	public void onItemClick(AdapterView parent, View view,
		int position, long id) {

		String type = getAusstellerByID(id).getType();

		if(type.equals("A")){
			openCompanyProfile(id);
		}else if(type.equals("C")){
			openMinimalProfile(id);
		}else if(type.equals("B")){
			openEducationalProfile(id);
		}
	}

	private void openCompanyProfile(long id){
		FragmentManager fm = fragment.getFragmentManager();
		fm.beginTransaction()
		.replace(R.id.container, CompanyProfileFragment.newInstance(1,id))
		.commit();
	}

	private void openMinimalProfile(long id){
		FragmentManager fm = fragment.getFragmentManager();
		fm.beginTransaction()
		.replace(R.id.container, MinimalProfileFragment.newInstance(1,id))
		.commit();
	}

	private void openEducationalProfile(long id){
		FragmentManager fm = fragment.getFragmentManager();
		fm.beginTransaction().replace(R.id.container,
		EducationalProfileFragment.newInstance(1,id))
		.commit();

	}

	...
}
```

Abb. 36 Ausschnitt aus der Implementierung der Klasse CompanyListListener

4.7 Ausstellersuche

Die Suche nach Ausstellern erfolgt entweder aufgrund der Standnummer oder des Ausstellernamens. Der Button für die Suche besitzt den *OnClickListener* `SearchListener`, der den Suchbegriff entgegennimmt und über *DAO*-Methoden entsprechende Aussteller findet. Wird kein Aussteller gefunden, so wird den Benutzer/innen eine Nachricht in Form eines modalen Fensters angezeigt. Andernfalls wird eine Instanz der Ausstellerliste erzeugt, welche nur die Aussteller beinhaltet, die dem Suchbegriff entsprechen. Die Implementierung dieser Funktionalität ist in Abb. 37 ersichtlich.

Wenn es sich bei der Eingabe des Suchbegriffes um einen Zahlenwert handelt, wird mit der Methode `getAusstellerByStandNr()` versucht, einen Aussteller aufgrund der Standnummer zu finden. Wenn es sich um einen alphanummerischen Wert handelt, wird mit der Methode `search()` nach Ausstellern gesucht, in deren Namen der Suchbegriff vorkommt.

```
public class SearchListener implements OnClickListener {
	//Attribute
	...

	@Override
	public void onClick(View v) {
		String searchString = input.getText().toString();

		if(isInteger(searchString)){
			results = new ArrayList<Aussteller>();
			results.add(getAusstellerByStandNr(searchString));
			if(results.get(0) == null){
				showSearchNotFoundAlert();
			} else {
				activity.getFragmentManager().beginTransaction()
				.add(AusstellerListFragment.newInstance(results, ...))

				.commit();
			}
		} else {
			List <String> strings = Arrays.asList(
			searchString.split(" "));
			results = search(strings);

			if (results.isEmpty()) {
				showSearchNotFoundAlert();
			} else {
				act.getFragmentManager().beginTransaction()
				.replace(R.id.container, AusstellerListFragment
				.newInstance(results, ...))
				.commit();
			}
		}
	}

	...
}
```

Abb. 37 Ausschnitt aus der Implementierung der Klasse SearchListener

4.8 Impulsvorträge/Messespecials

Die Implementierung der Liste von Impulsvorträgen und Messespecials verläuft analog zu den bereits vorgestellten Konzepten, die in der Ausstellerliste verwendet worden sind. Zuerst werden über entsprechende *DAO*-Klasse, alle Impulsvorträge oder Messespecials geladen, um daraufhin Referenzen auf die Listenlayouts zu erstellen. Des Weiteren werden Adapterklassen und *OnClickListener* erstellt. Innerhalb der Adapterklassen, werden die Layouts für die Listenelemente befüllt. Als Beispiel ist in Abb. 38 das *Fragment* für die Liste der Impulsvorträge zu sehen.

```
public class ImpulsvortraegeFragment extends Fragment {
      //Attribute
      ...

      @Override
      public View onCreateView(LayoutInflater inflater, ViewGroup
            container, Bundle savedInstanceState) {
            …
            rootView = inflater.inflate(
            R.layout.fragment_impulsvortraege, container, false);

            impDao = getDao();
            impDao.open();
            imp = iDao.getAllImpulsvortraege();
            impDao.close();

            list = (ListView)
            rootView.findViewById(R.id.ImpulsvortraegeList);

            ImpulsvortraegeListAdapter adapter = new
            ImpulsvortraegeListAdapter(rootView.getContext(), imp, this);

            list.setAdapter(adapter);
            list.setOnItemClickListener(new ImpListListener(this,
            rootView.getContext()));

            return rootView;
      }
      ...
}
```

Abb. 38 Ausschnitt aus der Implementierung der Klasse ImpulsvortraegeFragment

4.9 Lageplan

Um den Implementierungsaufwand in Grenzen zu halten, wurde bei der Realisierung des interaktiven Lageplans eine `WebView` eingesetzt. Bei einer `WebView` handelt es sich um eine `View`, die in der Lage ist, Webinhalte in Form von *HTML*-Dateien darzustellen. Dabei wird entweder über eine *URL* eine Webseite angefordert und mit der nativen Browser-Engine angezeigt oder eine lokale *HTML*-Datei aufgerufen. Innerhalb der `WebView` kann außerdem auch *JavaScript* ausgeführt werden.

Durch den Einsatz einer `WebView`, konnten viele Anforderungen des interaktiven Lageplans, mit relativ geringem Aufwand umgesetzt werden. Ein Beispiel hierfür ist die Zoom-Funktionalität. Viele Webseiten sind noch nicht für mobile Endgeräte optimiert. Um jedoch trotzdem eine adäquate Verwendung solcher Webseiten zu gewährleisten, bietet eine `WebView` standardmäßig die Möglichkeit, den Inhalt stufenfrei zu zoomen und zu verschieben. Würde der Lageplan mit einer `ImageView` innerhalb einer *Activity* implementiert worden sein, so müsste dieses Verhalten erst aufwendig implementiert werden.

Außerdem konnte mithilfe der *HTML*-Elemente `<map/>` und `<area/>` eine *OnClick*-Funktionalität, aufgrund der Positionen der Ausstellerstände am Lageplan, umgesetzt werden. In Abb. 39 ist ein Ausschnitt aus der *HTML*-Datei des Lageplans zu sehen. Hier gibt es zwei grundlegende Elemente.

Das erste Element `<div class="image">` beinhaltet zwei `<img/>`-Elemente. Das Erste stellt den Lageplan dar und das zweite Element ist ein Bild, mit denselben Dimensionen des Lageplans, welches nur aus einem einfarbigen Hintergrund besteht. Die Zwischenräume der Austellerstände am Bild des Lageplans sind transparent und somit wird bei allen Ständen der Hintergrund des Elementes `<img class="map_bottom"/>` dargestellt.

Diese etwas umständliche Repräsentation hat den Grund, dass wenn die Benutzer/innen beispielsweise von einem Ausstellerprofil *X* zum Lageplan wechseln, am Lageplan der Hintergrund des Standes des Ausstellers *X* mit einer anderen Farbe hervorgehoben wird. Es wird daher eine weitere Ebene mit der entsprechenden Farbe erstellt und zwischen dem Hintergrund und dem Lageplan gelegt. Nun besitzt Stand *X* einen roten Hintergrund und alle anderen Stände bleiben unverändert.

Das zweite Element `<map name="map_map">` definiert die Positionen am Lageplan, welche interaktives Verhalten aufweisen. Dieses Element beinhaltet für jeden Stand ein weiteres `<area/>`-Element, das die ID des Ausstellers und die Koordinaten am Lageplan definiert. Drücken die Benutzer/innen beispielsweise auf die Position `x:835 y:890` am Lageplan, so registriert das `<area/>`-Element mit der ID 1 diese Interaktion. Die IDs der `<area/>`-Elemente repräsentieren zugleich auch die IDs der zugehörigen Aussteller. Nach der Interaktion wird sofort die `onClick()`-Methode `showAusstellerPreview()` mittels *JavaScript* aufgerufen.

```
<html>
...

<body>
	<div class = "image">
		<img class="map_top" src="map.png" usemap="#map_map" />
		<img class ="map_bottom" src="map_underlay.png" />
	</div>
	<span class ="target" id="highlight"/>

	<map name="map_map">
		<area id="1" shape="rect" coords="840,1030,857,1047"
			onClick="showAusstellerPreview(id)">
		<area id="2" shape="rect" coords="770,1069,787,1086"
			onClick="showAusstellerPreview(id)">
		...
		<area id="36" shape="rect" coords="780,569,797,586"
			onClick="showAusstellerPreview(id)">
		<area id="37" shape="rect" coords="780,593,797,610"
		onClick="showAusstellerPreview(id)">
	</map>

	<script type="text/javascript">
		function showAusstellerPreview(id){...}
		...
	</script>
</body>
</html>
```

Abb. 39 Ausschnitt aus der Datei map.html

Die Funktion `showAusstellerPreview()` ruft mithilfe der ID und den Koordinaten des Standes die Methode `Android.showPreview()` auf. Bei dieser Methode handelt es sich um eine *Javascript*-Interface Methode. Das bedeutet, dass die Methode `showPreview()` nicht in *JavaScript*, sondern in *Java,* innerhalb eines zuvor definierten Interfaces implementiert ist. Es wird also innerhalb einer `WebView`, eine *Java*-Methode mittels *JavaScript* aufgerufen. Diese Kommunikation zwischen Java/Android und JavaScript/HTML ist notwendig, da im HTML-Code lediglich die IDs der Aussteller vorhanden sind und die Informationen zu den Ausstellern über die DAO-Schicht in der Applikation aufgerufen werden müssen.

Die beiden Funktionen `zoomToAussteller()` und `highlightAussteller()` stellen die umgekehrte Kommunikationsrichtung dar. Diese Methoden werden verwendet, um innerhalb einer *Activity* entsprechenden *JavaScript*-Code auszuführen. Sie werden aufgerufen, wenn beispielsweise in einem Ausstellerprofil der Lageplan-Button gedrückt wird. Jeder Aussteller wird mit seiner Position am Lageplan gespeichert und somit kann zu der entsprechenden Position am Lageplan gezoomt werden und anschließend der Stand mit der Methode `highlightAussteller()` hervorgehoben werden.

```
function showAusstellerPreview(id){
      var element = document.getElementById(id);
      var rect = element.getBoundingClientRect();

      Android.showPreview(id, rect.top , rect.left);
}

function zoomToAussteller(x,y){
      offsetX = screen.height/2;
      offsetY = window.innerHeight/2;
      scrollTo(x,y);
}

function highlightAussteller(x,y,n){
      var highlight = document.getElementById("highlight");
      highlight.style.border = "solid red " + n + "px";
      highlight.style.position = "absolute";
      highlight.style.top = y+6;
      highlight.style.left = x+6;
}
```

Abb. 40 Ausschnitt aus den JavaScript-Funktionen des interaktiven Lageplans

Die Implementierung des erwähnten JavaScript-Interfaces findet in der Klasse `MapInterface` statt und ist in Abb. 41 zu sehen. Wie in Abb. 40 zu sehen, wird die Methode `showPreview()` aufgerufen und ist dafür zuständig, nach dem Auswählen eines Standes ein Fenster zu öffnen, dass den Namen des Ausstellers und das Ausstellerlogo beinhaltet.

Dieses Vorschaufenster kann ebenfalls ausgewählt werden und die Benutzer/innen gelangen daraufhin zum Ausstellerprofil. Als Erstes wird in dieser Methode der Aussteller mithilfe der ID über einen *DAO*-Aufruf aus der Datenbank geladen. Danach wird das Layout des Vorschaufensters geladen, Name und Logo des Ausstellers in das Layout übertragen und ein *OnClickListener* für das gesamte Fenster registriert.

Zum Schluss wird ein `Toast`-Objekt erzeugt und das befüllte Layout wird diesem Objekt übergeben. Die Klasse `Toast` erzeugt ein modales Fenster, das innerhalb einer *Activity* angezeigt wird. Das Objekt wird der `MainActivity` übergeben und danach wird mit dem Aufruf `show()` das modale Fenster angezeigt.

```
public class MapInterface {
	//Attribute
	...

	public void showPreview(String id, String top , String left){

		this.ausstellerID = Integer.valueOf(id);
		Aussteller aussteller = getAusstellerByID(this.ausstellerID);


		View preview = inflater.inflate(R.layout.map_aussteller_preview,
		previewView.findViewById(R.id.toast_layout_root));

		ImageView logo = preview.findViewById(R.id.map_preview_logo);

		int resId = context.getResources().getIdentifier(
		aussteller.getLogo_name(), "drawable", context.getPackageName());

		logo.setBackgroundResource(resId);
		logo.setOnClickListener(new MapPreviewListener(aussteller.getId(),
		fragment, activity));

		TextView name = preview.findViewById(R.id.map_preview_name);
		name.setText(aussteller.getName());

		Toast toast = new Toast(context);
		toast.setView(preview);
		toast.setGravity(Gravity.TOP, 0, 124);
		act.setToast(toast);
		toast.show();

	}
	...
}
```

Abb. 41 Ausschnitt aus der Implementierung der Klasse MapInterface

In Abb. 42 ist ein Teil der Implementierung der Klasse `LageplanFragment` abgebildet. Dieses *Fragment* setzt die einzelnen Komponenten zur Realisierung des Lageplans zusammen. In der `onCreateView()`-Methode wird zuerst die `WebView` geladen. Mit dem Aufruf `setBuiltInZoonControls()` wird die Zoom-Funktionalität der `WebView` aktiviert und `setJavaScriptEnabled()` muss aufgerufen werden, um innerhalb der `WebView` *JavaScript*-Code auszuführen.

Die Methode `setWebViewClient()` wird nur aufgerufen, wenn ein Stand eines Ausstellers hervorgehoben werden soll. Der `WebViewClient` implementiert die Methode `onPageFinished()`, welche nach dem vollständigen Laden des HTML-Dokumentes ausgeführt wird. Hier wird der Aussteller mittels der ID geladen und die Koordinaten des Standes werden extrahiert. Anschließend wird durch den Methodenaufruf `loadUrl()` die JavaScript-Methode `zoomToAussteller()` aufgerufen, um mithilfe der Koordinaten den richtigen Stand zum Hervorheben zu finden.

In jedem Fall wird aber eine Instanz des *Javascript*-Interface `MapInterface` erstellt und der `WebView` übergeben. Es können beliebig viele *JavaScript*-Interfaces verwendet und der `WebView` zugewiesen werden. Jedes Interface wird dabei mit einem eigenen Schlüsselwort angesprochen. Im Fall des Lageplans existiert nur ein *JavaScript*-Interface, welches durch den Aufruf `addJavascriptInterface()` und dem Schlüsselwort `Android` registriert wird. Zum Schluss wird die lokale *HTML*-Datei `map.html` geladen und es wird das vollständige `WebView`-Objekt zurückgeliefert.

```
public class LageplanFragment extends Fragment {
	//Attribute
	...
	@Override
	public View onCreateView(LayoutInflater inflater, ViewGroup container,
			Bundle savedInstanceState) {
		...

		map = (WebView) rootView.findViewById(R.id.webViewMap);
		map.getSettings().setBuiltInZoomControls(true);
		map.getSettings().setJavaScriptEnabled(true);

		...

		//wird nur Aufgerufen, wenn ein Stand hervorgehoben werden soll
		map.setWebViewClient(new WebViewClient(){
			@Override
			public void onPageFinished(WebView view, String url) {
				super.onPageFinished(view, url);


				aus = getAusstellerById(id);
				int x = aus.getStand().getX1();
				int y = aus.getStand().getY1();

				map.loadUrl("javascript:zoomToAussteller(" + x + ","
				+ y +")");
			}
		});

		MapInterface jsInterface = new MapInterface(rootView.getContext(),
		rootView, (MainActivity)getActivity(), this);

		map.addJavascriptInterface(jsInterface, "Android");
		map.loadUrl("file:///android_asset/map/map.html");

		return rootView;
	}
	...
}
```

Abb. 42 Ausschnitt aus der Implementierung der Klasse LageplanFragment

Um die Koordinaten der einzelnen Ausstellerstände zu ermitteln, wurde ein eigenes Programm erstellt (*Map Area Tool*). Das Programm benötigt lediglich das Bild des Lageplans, in denselben Dimensionen, wie es mit der Applikation ausgeliefert wird. In Abb. 43 ist die einfache Funktionsweise des Programmes ersichtlich. Es zeigt den Lageplan an und durch das Klicken auf eine Position wird ein modales Fenster geöffnet.

In diesem Fenster trägt man die ID des Ausstellers und die Standnummer ein und bestätigt die Eingabe. Daraufhin wird der Bereich den man ausgewählt hat mit rot markiert. Beim Klicken wird immer die obere linke Ecke als Referenzpunkt angenommen.

Somit kann man die Positionen sämtlicher Stände ermitteln und anschließend durch einen Klick auf die Export-Funktion, die Positionen aller Stände exportieren. Beim Exportieren werden die Informationen in Form des benötigten HTML-Codes in einer Datei gespeichert. Das bedeutet, dass das gesamte `<map/>`-Element vollständig in der exportierten Datei vorliegt und nur noch in die Datei `map.html` kopiert werden muss.

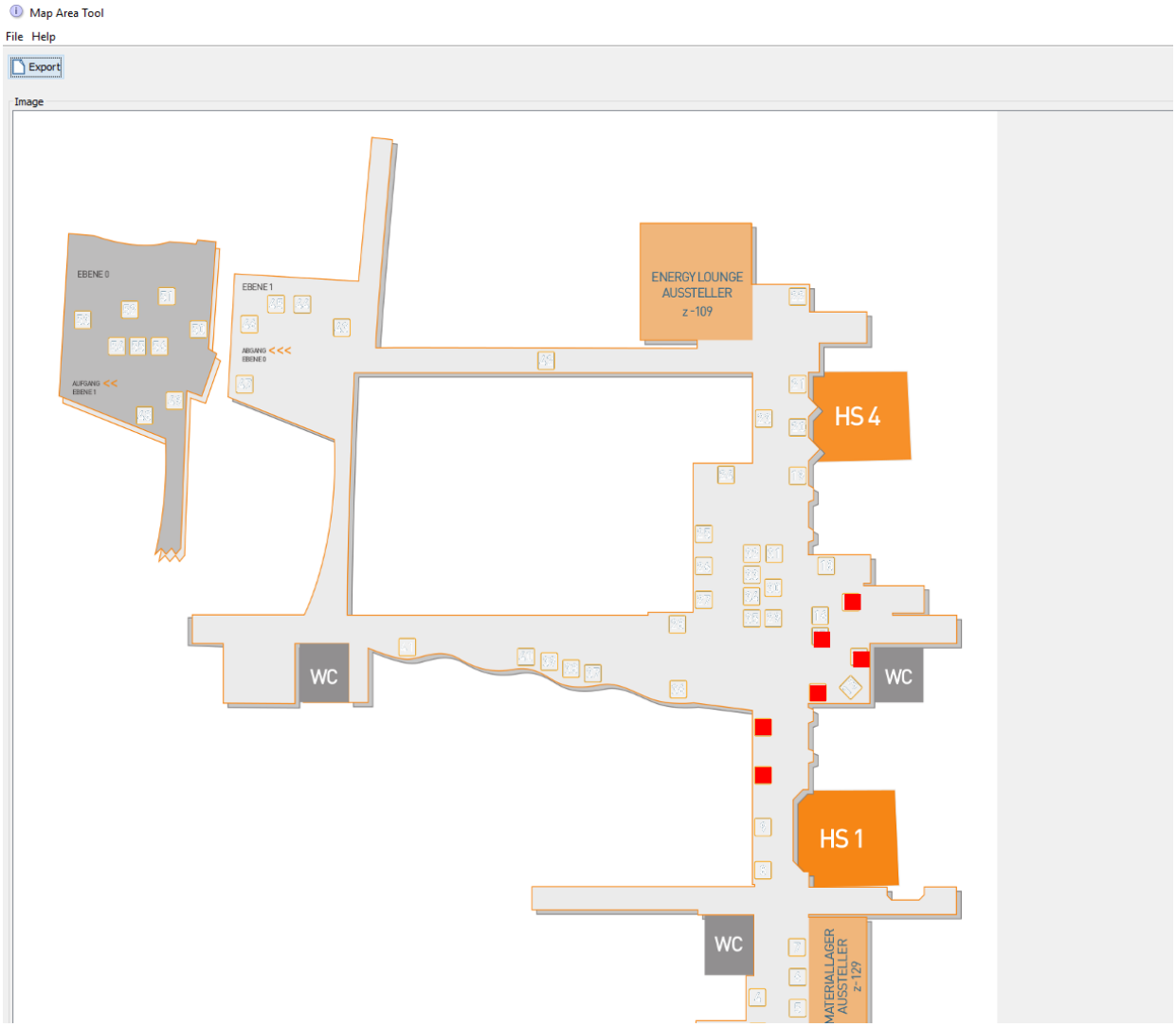

Abb. 43 Map Area Tool

4.10 Datenmigration

Alle textuellen Informationen der Messe-App (Aussteller, Messespecials, Impulsvorträge, etc.) sind in Form mehrerer Excel-Dokumente vorhanden. Diese Daten müssen somit zuerst vorbearbeitet werden, um anschließen über die Datenbank der Applikation aufgerufen werden zu können. Die einfachste Möglichkeit wäre, sämtliche *DDL*- und *DML*-Statements[12] aufgrund der Informationen in den Excel Tabellen per Hand zu verfassen. Das würde jedoch auch bedeuten, dass dieser zeitaufwendige und ineffiziente Vorgang jedes Jahr wiederholt werden müsste.

Um einen schnelleren Weg der Datenmigration zu gewährleisten, wurde ebenfalls ein eigenständiges Programm entwickelt, dass in der Lage ist, aus den Excel-Tabellen, fertige und vollständige *DDL*- und *DML*-Statements zu exportieren.

Es handelt sich dabei um eine *Java*-Anwendung, die Zugriff auf die Tabelleninformation in Form von *CSV*[13]-Dateien benötigt. Die Anwendung erstellt zuerst alle benötigten *DDL*-Statements für die Datenbank. Diese DDL-Statements sind statisch und sollten sich über mehrere Jahre hinweg nicht ändern. Die Statements werden in eine Textdatei geschrieben und können danach zur Erstellung der Datenbank herangezogen werden. Die *DML*-Statement werden aufgrund der Informationen in den *CSV*-Dateien erzeugt und werden ebenfalls in eine Textdatei geschrieben.

Somit beschränkt sich der jährliche Aufwand für die textuelle Datenmigration, auf das Speichern der Excel-Tabellen als *CSV*-Dateien und das Ausführen des Migrationsprogrammes. Danach kann die Datenbank mit den *DDL*- und *DML*-Statements erstellt und in das Android-Projekt übernommen werden.

Der letzte Schritt in der Datenmigration ist das Bearbeiten der Ausstellerlogos und der Bilder der Ansprechpersonen. Diese Bilder werden von den Ausstellern geliefert und liegen in den unterschiedlichsten Bildformaten und Dimensionen vor. Die Connect-App benötigt jedoch alle Bilddateien im *JPEG*- oder *PNG*-Format. Logos müssen außerdem ein bestimmtes Seitenverhältnis aufweisen, um eine konsistente Darstellung in der Listenansicht und in den Ausstellerprofilen zu gewährleisten. Für die Bilder der Ansprechpersonen ist eine fixe Größe vorgegeben.

Daher müssen jedes Jahr, die Logos und Fotos der Ansprechpersonen freigestellt und gegebenenfalls in ein korrektes Dateiformat exportiert werden, bevor sie in die App übernommen werden können.

4.11 Veröffentlichung

Nachdem der Vorgang der Datenmigration abgeschlossen ist und alle benötigten Informationen in den richtigen Formaten in der Applikation vorliegen, kann die App zur Veröffentlichung in den *Google Play Store* hochgeladen werden. Zuvor müssen der Programmcode und die Daten jedoch noch in ein einziges Paket verpackt werden. Dieses Paket wird schlussendlich in den *Play Store* hochgeladen und von den Benutzer/innen heruntergeladen und installiert.

Bei dem Paket handelt es sich um eine *APK*(*Android Application Package*)-Datei. Eine *APK*-Datei liegt in einem spezifischen Containerformat vor und beinhaltet unter anderem Metainformationen, Signaturen, kompilierten Quellcode und statische Ressourcen. Diese APK-Datei kann mithilfe von entsprechenden Tools erstellt und signiert werden. Eine Signatur muss für jede Applikation einmalig erstellt werden und wird mit der APK-Datei ausgeliefert.

Einer der einfachsten Wege eine APK-Datei zu erstellen, ist mithilfe der Entwicklungsumgebung *Android-Studio*. Diese bietet eine Funktion zum Erstellen einer signierten APK-Datei an. Die fertige APK-Datei muss nun im *Google Play Store* hochgeladen werden (Abb. 44).

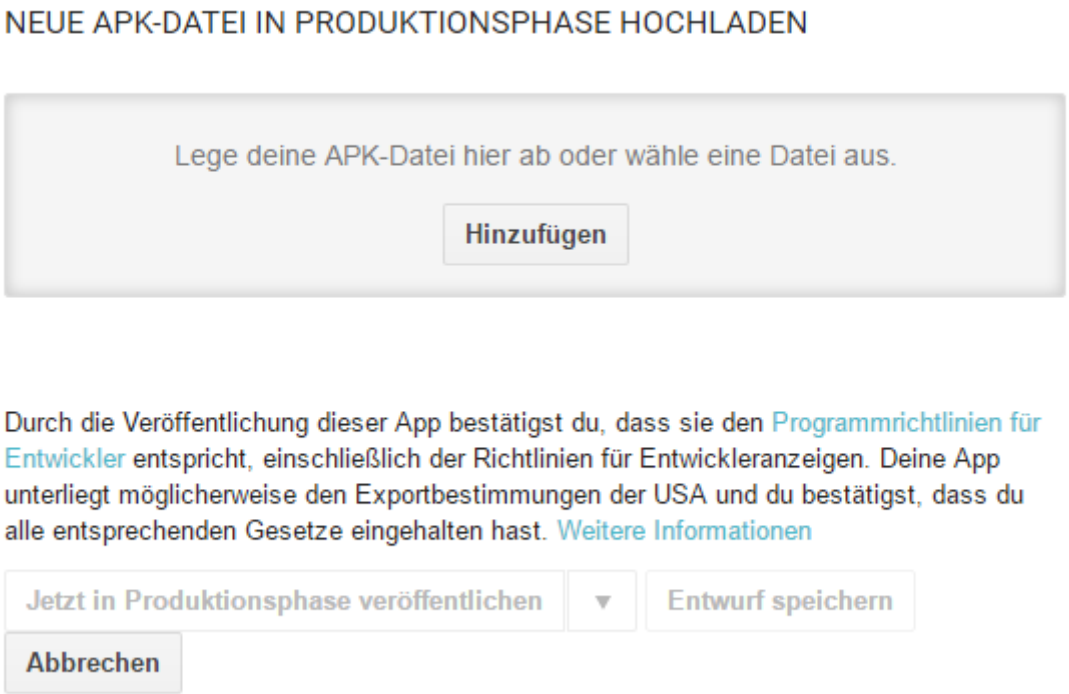

Abb. 44 Hochladen der APK-Datei im Google Play Store

Danach müssen noch einmalig Beschreibung, Funktionsgrafiken und Screenshots der Applikation hochgeladen werden, welche im *Play Store*-Eintrag angezeigt werden. Sind alle Voraussetzungen zur Veröffentlichung der App erfüllt und die Veröffentlichung im Play Store bestätigt, so dauert es in der Regel einige Stunden bis Tage, bis die Applikation weltweit verfügbar ist.

4.12 Updates

Da sich jedes Jahr die Informationen der Aussteller, Impulsvorträge und Messespecials sowie die Stände am Lageplan ändern, muss auch jedes Jahr ein Update der Connect-App vorgenommen werden. Dazu wird der bereits beschriebene Datenmigrationsprozess durchgeführt und anschließend eine neue *APK*-Datei erstellt. Zu beachten ist hierbei, dass man die *APK*-Datei mit der Signatur der ersten Version signieren muss. Wird diese Signatur unglücklicherweise verloren, so ist es nicht mehr möglich, ein Update der Applikation im *Play Store* durchzuführen.

Außerdem müssen vor dem Verpacken die Versionscodes und Versionsnamen in der Datei `AndroidManifest.xml` aktualisiert werden. Üblicherweise startet die erste Version einer App mit dem Versionscode *1* und dem Versionsnamen *1.0*. Nach jedem Update wird der Versionscode um eins erhöht und der Versionsname ändert sich ausgehend von dem Umfang der Änderungen und Neuerungen auf *1.0.1*, *1.1*, oder *2.0*.

Die Erhöhung des Versionsnamens auf *1.0.1,* weist auf lediglich kleine Änderungen in der Applikation hin. Eine Änderung auf *1.1* auf weitreichendere Änderungen und eventuell neue Funktionalität. Beim Erhöhen der Vorkommastelle, spricht man von einem *Major-Update,* welches auf viele Änderungen, Verbesserungen und neue Funktionen hinweist.

Die Versionscodes sind für Benutzer/innen nicht ersichtlich und müssen bei jedem Update in fortlaufender Reihenfolge vergeben werden. Versionsnamen unterliegen lediglich der oben beschriebenen Konventionen, sind von den Benutzern/Benutzerinnen sichtbar und werden vom *Play Store* nicht in Hinblick auf fortlaufende Nummern validiert.

Ist ein Update in Form einer neuen APK-Datei im Play Store hochgeladen, so gibt es zwei Möglichkeiten, wie Benutzer/innen dieses Update erhalten können. Die erste Möglichkeit besteht darin im *Play Store*-Eintrag auf den Button *Aktualisieren* zu drücken. Die Zweite Möglichkeit ist im Zuge eines regelmäßigen Updates, welches vom Android-Betriebssystem durchgeführt wird und für alle installierten Apps nach verfügbaren Updates sucht und diese installiert.

5. Zusammenfassung

In der sechsmonatigen Entwicklungsphase der Connect-App, wurde von allen beteiligten Personen, hohe Kooperationsbereitschaft und Engagement gezeigt.

Somit konnten alle entstehenden Herausforderungen und Probleme rasch bewältigt werden und pünktlich zum Release-Termin, eine den Anforderungen entsprechende Applikation, ausgeliefert werden. Ein weiterer Faktor, der die Entwicklung positiv beeinflusst hat, ist die Tatsache, dass alle androidspezifischen Konzepte und Technologien hervorragend dokumentiert sind. Das Endergebnis dieses Projektes ist eine App, die über mehrere Jahre im Google *Play Store* veröffentlicht wurde und während dieser Zeit keinen Absturz gemeldet hat.

Im Zuge der Entwicklung und vor allem bei der jährlichen Datenmigration sind jedoch auch einige Punkte ersichtlich geworden, die effizienter und besser gelöst werden hätten können.

Eine der ersten Entscheidungen, die bei der Planung der Connect-App getroffen wurden, war es, dass die App ohne Internetverbindung funktionieren sollte. Das hat zum einen den Vorteil, dass die Benutzer/innen die App, unabhängig von der mobilen Datenübertragung nutzen können, zum anderen jedoch auch einige Nachteile.

Es muss jedes Jahr ein Update der App hochgeladen werden, damit die Benutzer/innen die aktuellen Informationen in der App einsehen können. Dieses Update kann jedoch nur von Personen durchgeführt werden, die ein grundlegendes Verständnis von Programmierung, Datenbanken und Bildbearbeitung haben.

Ein möglicher Lösungsansatz wäre hier die Informationen der App auf einem Webserver bereitzustellen und die App mit diesem Webserver kommunizieren zu lassen. Die Benutzer/innen müssten sich somit nicht um ein jährliches Update kümmern und hätten permanent die aktuellsten Informationen. Das Problem der Datenmigration könnte mithilfe eines *Content Management Systems* (*CMS*) gelöst werden.

Dieses *CMS* kann auf einem Webserver bereitgestellt werden und bietet, den für den Inhalt verantwortlichen Personen, Funktionen wie das Anlegen neuer Aussteller, Impulsvorträge, und Messespecials. Außerdem kann damit das Produktiv- und Offlinestellen ganzer Messeinhalte realisiert werden. Alle Informationen werden in einer Datenbank gespeichert und über eine definierte *API*[14]-Schnittstelle kann die App die Daten abrufen.

Dieser Lösungsansatz ist jedoch um einiges komplexer und umfangreicher als die derzeitige lokale Variante der Connect-App und hätte wahrscheinlich in der vorgegebenen Zeit und mit den vorhandenen Ressourcen nicht umgesetzt werden können. Zusätzlich würden auch regelmäßige Kosten für Serverinfrastruktur und Wartung entstehen.

Das Navigationskonzept ist ein weiterer verbesserungsbedürftiger Aspekt. Es stellte sich zum Ende hin sehr klar heraus, dass einige Benutzer/innen das Konzept des *NavigationDrawer* nur schwer nachvollziehen konnten und teilweise lange nach den Menüpunkten suchen mussten. Hier wäre wahrscheinlich eine Navigation in Form von Tabs oder Icons benutzerfreundlicher gewesen.

Obwohl es einiges an Verbesserungsbedarf in der aktuellen Version der Connect-App gibt, ist das Feedback der Benutzer/innen überwiegend positiv ausgefallen. Somit konnte auch ein erfolgreicher und positiver Abschluss des Projektes erzielt werden. Die oben genannten Verbesserungen und das *CMS*-Konzept, sind sicherlich ein guter Wegweiser und Startpunkt für eine Connect-App 2.0.

6. Referenzen

1. Im *Google Play Store* werden Applikationen für Android-Geräte veröffentlicht und die Benutzer/innen können diese Apps an einem zentralen Ort herunterladen.

2. https://de.wikipedia.org/wiki/Prototyping_(Softwareentwicklung) 01.09.2015

3. Als *Mockup* wird ein Modell, das eine Nachbildung einer Entität darstellt, bezeichnet. Es wird meist zu Präsentationszwecken verwendet.

4. Als *Usability* wird der Grad der Einfachheit und intuitiven Bedienung einer Anwendung bezeichnet.

5. *XML* (Extensible Markup Language) ist eine erweiterbare Auszeichnungssprache die in der Softwareentwicklung häufig zum Einsatz kommt.

6. Die Abbildung wurde aus der offiziellen Android-Dokumentation entnommen: https://developer.android.com/guide/components/fragments.html 17.11.2016

7. Die Abbildung wurde aus der offiziellen Android-Dokumentation entnommen: https://developer.android.com/training/basics/activity-lifecycle/starting.html - 17.11.2016

8. Der Programmcode wurde aus der offiziellen Android-Dokumentation entnommen: https://developer.android.com/guide/topics/data/data-storage.html#pref – 20.11.2016

9. *MySQL* und *OracleDB* sind eine der marktführenden relationalen Datenbanken die sowohl im kommerziellen, als auch in privaten Bereichen genützt werden.

10. *Design-Guides* sind Richtlinien und Konventionen, die von Entwickler/innen eingehalten werden sollen, um wartbare und verständliche Software zu erstellen.

11. Methoden die nur dem Zweck dienen, ein Objekt zurückzuliefern, werden als *Getter-Methoden* bezeichnet. Methoden mit denen man nur *Attribute* zuweist, werden als Getter-Methoden bezeichnet.

12. *DDL*-Statements sind Datenbankbefehle, mit denen Objekte erzeugt werden. *DML*-Statements sind Datenbankbefehle mit denen Objekte verändert werden.

13. *CSV* steht für *Comma Separated Values* und beschreibt ein Dateiformat, indem Werte durch Beistriche und Semikolons getrennt werden.

14. *API* (*Applikation Programming Interface*) bezeichnet eine Schnittstelle zur Anwendungsprogrammierung. Eine *API* wird von einem Softwaresystem zur Verfügung gestellt und ermöglicht es anderen Systemen damit in standardisierter Form zu kommunizieren.