



Business Informatics

Bachelor's Thesis

Designing and Developing an Android Application for a Mobile Computing Course

presented by: Julian Döring

1. Reviewer: Prof. Dr. Andreas Winter
2. Reviewer: Dipl.-Inform. Dirk Peters

Oldenburg, October 30, 2013

Contents

List of Figures	6
List of Tables	6
List of Listings	9
1 Introduction	10
1.1 Objective	10
1.2 Approach	11
1.3 Structure	12
2 Course Framework	13
2.1 Description of the Four Iterations	13
2.2 Qualification of the Course Participants	14
3 Tool Environment	16
3.1 Development Environment	16
3.2 GUI Builder	17
3.3 Emulator	18
3.4 Tool Set Up	19
3.4.1 ADT-Bundle	20
3.4.2 Emulator	20
3.4.3 Connecting an Android Device	21
4 Application Requirements	22
4.1 Application Vision	22
4.2 Similar Application Africa Life	25
4.3 Use Cases	26
4.4 User Stories	28
4.4.1 Add a Sighting	29
4.4.2 View all Sightings	31
4.4.3 View map	32
4.4.4 Share a Sighting	33
5 Architecture	34
5.1 Model-View-Presenter	34

5.2	WildlifeAfrica Architecture	35
6	Iteration 1: Android Framework	37
6.1	Hello World Application	37
6.2	Project Structure	37
6.3	Manifest File	38
6.4	Resources	38
6.5	Components	39
6.6	SQLite Database	39
6.7	WildlifeAfrica Project	39
6.7.1	Project Setup	40
6.7.2	Download Google Play SDK	40
6.7.3	Install Google Play SDK	40
6.7.4	Reference Google Play SDK	40
6.8	Time	41
7	Iteration 2: GUI	42
7.1	Navigation and Screens	42
7.2	Mock-Up	45
7.3	Android Manifest	47
7.4	Resource Files	47
7.5	Activities	48
7.5.1	Action Bar	48
7.6	Activity Layout Files	52
7.7	AddSightingActivity	53
7.8	SightingListActivity	56
7.9	SightingMapActivity	63
7.10	SightingDetailsActivity	64
7.11	SightingFilterActivity	64
7.12	Sighting	66
7.13	Time	67
8	Iteration 3: Database and Sensors	69
8.1	Android Manifest	69
8.2	Database Layer	70
8.2.1	SightingsDBHelper and NewSightingDBHelper	71
8.2.2	SightingsDatabase	72

8.3	Database Layer Usage	78
8.3.1	AddSightingActivity	78
8.3.2	SightingListActivity	79
8.3.3	SightingMapActivity	81
8.3.4	SightingDetailsActivity	81
8.4	ImageHelper	82
8.4.1	Usage	83
8.5	Camera	83
8.6	Gallery	85
8.7	LocationService	86
8.8	LocationService Usage	87
8.8.1	Android Manifest	88
8.8.2	Start Location Service	88
8.8.3	Broadcast Receiver	88
8.8.4	Manage Broadcast Receiver	90
8.9	Internet	91
8.10	Time	91
9	Iteration 4: External Services	93
9.1	External Database	93
9.2	GoogleMaps	96
9.2.1	Manifest	96
9.2.2	Layout File	97
9.2.3	Initialize the Map	97
9.2.4	Camera and Position	98
9.2.5	Marker on the Map	98
9.2.6	OnInfoWindowClickListener	99
9.3	Sharing	99
9.4	SyncService	100
9.4.1	IUpdateSightingDatabase	101
9.4.2	UploadService	102
9.4.3	DownloadService	106
9.4.4	BroadcastReceiver	109
9.4.5	Android Manifest	110
9.5	Time	111

10 Conclusion	112
11 Outlook	114
12 CD Content	115
13 List of References	116

List of Figures

1	Geanymotion	20
2	Detailed use case diagram	27
3	Overview of the main use cases	28
4	Model-View-Presenter pattern	34
5	Application screen structure	36
6	Android project structure	38
7	Application screen structure	43
8	Structure of the screens	44
9	Mock-up: sightings in a list (left) and on a map (right)	45
10	Mock-up: filter sightings (left) and sighting details (right)	46
11	Mock-up: new sighting	46
12	Mock-up: legend for the icons	46
13	Action Bar for the SightingListActivity	48
14	Action Bar for the SightingMapActivity	51
15	Action Bar for the AddSightingActivity	51
16	Action Bar for the SightingFilterActivity	51
17	Action Bar for the SightingDetailsActivity	51
18	Icon that indicates a picture	53
19	Classdiagram: Sighting class	67
20	Classdiagram: Database	70
21	Classdiagram: LocationService	86
22	Folder structure on the cd	115

List of Tables

1	User Story: Add a Sighting	29
2	User Story: View all sightings	31
3	User Story: View map	32
4	User Story: Share a sighting	33
5	Icons for the action bar	52
6	MySQL Database: Table	94

Listings

1	Manifest: register activities	47
2	Resources: String-Array	47
3	Action Bar: Layout SightingListActivity	49
4	Action Bar: set the layout	49
5	Action Bar: react to click	50
6	Action Bar: disable title	50
7	Action Bar: enable Home as up	51
8	Action Bar: react to click on the application icon	51
9	AddSightingActivity: instance variables	53
10	AddSightingActivity: onCreate() method	54
11	AddSightingActivity: backward navigation	55
12	SightingListActivity: instance and class variables	56
13	SightingListActivity: navigation in the action bar	57
14	SightingListActivity: onActivityResult() method	57
15	SightingListAdapter: implementation	58
16	SightingListActivity: use SightingListAdapter	60
17	SightingListActivity: Spinner OnItemSelectedListener	61
18	SightingListActivity: SightingNameReverseComparator	62
19	SightingListActivity: sortSightingsZToA() function	62
20	SightingListActivity: onResume() method	63
21	Layout file: activity_sighting_map.xml	63
22	SightingDetailsActivity: set up GUI elements	64
23	SightingFilterActivity: Result OK	65
24	SightingFilterActivity: Result CANCEL	65
25	FilterListAdapter: layout details	66
26	AndroidManifest: Permissions to Sensors and Storage	69
27	SightingsDBHelper: column names and create statement	71
28	SightingsDBHelper: methods	72
29	SightingDatabase: open and close the database	73
30	SightingsDatabase: addSighting	74
31	SightingDatabase: data into ContentValues	74
32	SightingsDatabase: add sighting information	74
33	SightingsDatabase: getSighting	75
34	SightingsDatabase: getSightingList()	76

35	SightingDatabase: create a list with sightings	76
36	SightingsDatabase: getFilteredSightingList()	77
37	AddSightingActivity: SimpleDateFormat	78
38	AddSightingActivity: expand saveSighting()	79
39	SightingListActivity: getSightingList()	80
40	SightingListActivity: expand onResume()	80
41	SightingListActivity: expand onActivityResult	80
42	SightingDetailsActivity: get sighting data	81
43	Intent to start SightingDetailsActivity	82
44	ImageHelper: implementation	82
45	SightingListAdapter: show image	83
46	AddSightingActivity: start the camera app	84
47	AddSightingActivity: CAMERA_REQUEST	84
48	Handle the result from camera request	84
49	Intent to start the gallery app	85
50	AddSightingActivity: gallery request	85
51	Handle the result from gallery request	86
52	LocationService: ask for location data	87
53	LocationService: receive location data	87
54	Manifest: register LocationService	88
55	Manifest: intent-filter	88
56	SightingListActivity: start location Service	88
57	SightingListActivity: add a BroadcastReceiver	89
58	SightingListActivity: setLocation()	89
59	SightingListActivity: calcDistanceForSightings()	89
60	SightingListActivity: register the BroadcastReceiver	90
61	SightingListActivity: deregister the BroadcastReceiver	90
62	Check if internet connection is available	91
63	PHP script: addSighting.php	94
64	PHP script: getAllSightings.php	95
65	Manifest: permission for maps	96
66	Manifest: OpenGL ES	96
67	Manifest: Google Maps	97
68	SightingMap Layout: Fragment for the Map	97
69	SightingMapActivity: initialize the map	97
70	SightingMapActivity: show own position	98

71	SightingMapActivity: setMarkerOnMap()	98
72	SightingMapActivity: OnInfoWindowClickListener	99
73	SightingDetailsActivity: share functionality	100
74	SightingDatabase: getSightingsFromNewSightingDB()	101
75	SightingDatabase: updateSightingDatabase()	102
76	SightingDatabase: deleteAllSightings	102
77	SightingDatabase: clearNewSightingsDatabase()	102
78	UploadService: variables	103
79	UploadService: methods	103
80	DownloadService: variables	106
81	DownloadService: methods	107
82	SightingMapActivity: create the BroadcastReceiver	109
83	SightingListActivity: create BroadcastReceiver	110
84	SightingMapActivity: register the BroadcastReceiver	110
85	SightingMapActivity: deregister the BroadcastReceiver	110
86	Manifest: UploadService and DownloadService	110
87	Manifest: intent-filter	111

1 Introduction

The created bachelor's thesis is a collaboration between the chairs of the VLBA (Very Large Business Applications) and the Software Engineering Group at the Carl von Ossietzky University Oldenburg within the DASIK Project.

The main goal of the *DASIK*-Project [das13] is to offer intensive courses on several topics of information and communication technologies both for students from academic institutions and for employees from industry. Sponsors of the *DASIK*-Project are the German Academic Exchange Service (DAAD) [DAA13] and the Federal Ministry for Economic Cooperation and Development (BMZ) [BMZ13]. In order to find topics for the courses, several industry partners were involved to provide practical relevance. The courses are created by the consortium consisting of the University Oldenburg (Germany), the Nelson Mandela Metropolitan University (South Africa) and industrial partners.

A one week intensive course with the topic of *mobile computing* is currently being planned. The goal of the course is to convey basic concepts of mobile mobile designing, developing and evaluating mobile applications. This will be consolidate with a practical Android application.

Time is tight in an intensive course and so it is difficult to develop a complete Android application in the practical part of the intensive course. Therefore, the course needs an application, which covers the conceptual contents and can be used as a basis for the practical part. The application thus to provide a framework that can be extended by the modules during the teaching units. It also provides sample solutions for the different modules.

1.1 Objective

The main goal of this bachelor's thesis is to provide a *mobile application* for the mobile computing course. The mobile computing course imparts knowledge in a theoretical part about designing and evaluating a mobile application for different platforms. The practical part of the course will be to develop an Android application ([Mei12], [HB12]). Therefore, this bachelor thesis deals with the conceptual design and implementation of a mobile Android application that can be extended modularly. The functionality of this Android application is based on the iterations of the intensive course, which are defined in an internal document [WCW⁺13]. The course will be a one week intensive course and the topics for the iterations are defined as follows:

Mo: Introduction to the development environment and the Android framework

Tu: Designing and building a graphical user interface

We: Using databases and sensors

Th: Integration of external services (e. g. Flickr or Facebook)

Modularity of the application and documentation of the application are important, so that the course participants can reconstruct the certain parts, in case they are not able to create them by their own.

This mobile computing course will taking place at the Nelson Mandela Metropolitan University in South Africa, so that the scenario for this application gets a local context. Near the Nelson Mandela Metropolitan University in Port Elizabeth is the Kragga Kamma Game Park, which is the home of a lot of animals. The application should enable visitors of this Game Park, or an other Game Park, to document his animal sightings, to review or share them with others. The applications displays the sightings with the animal name, a timestamp and a photo, if available. Furthermore the application contains a map to show the location of a sighting.

The application will show a map of the Kragga Kamma Game Park or the place where the user is. It displays the last encounters, timestamp and location with animals and the user can add new encounters. In addition the user can share the encounters with his his family and friends using external services like Flickr or Facebook.

1.2 Approach

To build this application, requirements are needed. These requirements are elicited based on the contents of the mobile computing course. The course is divided into several topics and the implementation of the application happens in iterations. Each iterations is related to one topic of the course.

The first iteration will cover a fundamental framework, so that an overarching structure and functionality will be provided. The following iterations are aligned with the course topics to make sure that the contents are all covered. The outcome of each iteration is a runnable increment of the application.

The next step will be the design and implementation of a graphical user interface, which includes the navigation within the Application. At the end of this iteration a first runnable increment of the application is created, that shows the graphical user interface of the application.

Afterwards, a database is provided and the interfaces defined and implemented. Also the functionality of camera and the GPS module are added to the application. This increment

can access and use different sensors and also read and write data to a database.

In the last iteration, external services are integrated. This covers external services like Twitter or Facebook, the Google Maps service as well as access to an external server to synchronize the local database with the database on the server.

1.3 Structure

This bachelor's thesis is structured as follows. Chapter 2 describes the four iterations of the mobile computing course and the prerequisites, which the course participants have to comply. Chapter 3 introduces tools, that are used to develop the Android application. For each tool several alternatives are presented, before a decision is made. This chapter describes also how to setup these tools. Chapter 4 describes the software requirements for this application. It provides an application vision, use cases, user stories and a list of requirements, to concretize the application. Chapter 5 describes the architecture for this application. The implementation of this application follows the iterations of the course. At the end of each iteration a new increment of the application is developed, the following four chapters describes the implementation. Chapter 6 ("Android Framework") introduces key components of the Android SDK and the Android project for the following iterations is prepared. The implementation of the iteration "GUI" is described in chapter 7, iteration "Databases and Sensors" in chapter 8 and iteration "External Services" in chapter 9. Each of this four chapters ends with section, that describes how long the author has needed to complete the iteration. This is also the validation, that the application can be developed in this iterations. Finally, chapter 10 is the conclusion of this bachelor's thesis. This follows an outlook, chapter 11, and a description of the CD content, chapter 12.

2 Course Framework

The first section of this chapter describes the four iterations that the course participants will pass through within this mobile course (Section 2.1). To make sure that all course participants have the same knowledge base, qualifications on the course participants are introduced (Section 2.2).

2.1 Description of the Four Iterations

The goal of this mobile computing course is to impart knowledge about development for mobile devices especially for Android. The course is divided into four general topics. Each of these topics has its own learning outcomes. The topics are:

- Framework
- Graphical User Interface
- Databases and Sensors
- External Services

Each day of the course is divided into a theoretical part in the morning and a practical part in the afternoon. The learning outcomes in the morning are the required learning incomes for the practical part in the afternoon.

This learning outcomes are defined in an internal document [WCW⁺13] and are described in the following sections.

Framework

The topics of the first day are an introduction to mobile application platforms and development tools, an introduction to Android design principles and development for Android in general as well as the Android SDK. The theoretical part covers topics like the Android architecture and an introduction to Eclipse. During the practical part, the participants will setup Eclipse as development environment and they will write their first “Hello World”-Application and deploy it.

Furthermore, the scenario for the application will be presented and the course participants will collect general requirements. For the learning outcomes of this day, the participants know about components and characteristics of mobile application and they are able to develop minimal Android applications with Eclipse. The participants will know how to set up an Android Project in Eclipse and how to deploy an application onto a real device.

Graphical User Interface

In this iteration, the theoretical part provides knowledge about Mobile Design Principles, UI Design Patterns and the usage of Mobile UI design tools. In the practical part, the graphical user interface for the application is designed and built.

It is necessary to build the complete user interface in this iteration, because in the following iterations requirements occur, which affect the user interface. The complete user interface helps in further iterations to concentrate on the existing tasks.

The learning outcomes from this day are the ability to assess and apply mobile design principles and UI design patterns for the theoretical part as well as UI Mock-ups of mobile UI for the practical part. In addition the participants developed a first increment of the GUI for their application.

Databases and Sensors

This iteration covers the topics SQLite databases in Android and database-binding by a database interface in Android. The usage of sensors in Android like the camera module or the GPS module is also covered. All data which can be collected by the sensors are stored in a database.

The learning outcomes from this iteration are the use of databases and sensors. The application supports this learning outcomes by providing a database schema and interfaces to use an existing database.

External Services

In this iteration, external services are added to the application. The participants learn how internet resources are used for Android applications. During the practical part, the application is extended through external services. Examples are the integration of Google Maps or services like Facebook and Flickr.

The learning outcomes of this iteration are the use of external services. The application supports this by providing all needed libraries and interfaces for easy usage.

2.2 Qualification of the Course Participants

This section describes the skills that are required for this course. On the one hand, interested people can decide whether they can improve their skill with this course. On the other hand prerequisites make sure that all participants share minimal skills to cope with

the course content. To make this prerequisites verifiable, they are referred to courses at the University of Oldenburg.

Prerequisites for this mobile computing course are:

1. The course participants must have pass the Java Programming Course [OUO13d].
2. The course participants must have passed the course 'Algorithmen und Programmierung' [OUO13b].
3. The course participants must have passed the course 'Algorithmen und Programmierung' [OUO13b].
4. The course participants must have passed the course 'Algorithmen und Datenstrukturen' [OUO13a].
5. The course participants must have passed the course 'Algorithmen und Datenstrukturen' [OUO13a].
6. The course participants must have passed the course 'Softwaretechnik 1' [OUO13e].
7. The course participants must have passed the course 'Softwaretechnik 1' [OUO13e].
8. The course participants must have passed the course 'Informationssysteme 1' [OUO13c].

3 Tool Environment

Using tools to develop an Android application can simplify and accelerate the developing process. For Android development several tools for several tasks are available. This chapter describes the tools that are proposed for development and the requirements for these tools, which are needed for the mobile computing course. This tools will support the development process in all phases from the project set up to deploying an application on a device.

For each needed, tool requirements are elicited and the advantages and disadvantages of the several alternative tools are listed. The tools that are needed are an IDE (Integrated Development Environment, see section (Section 3.1), a GUI Builder (Section 3.2), an Emulator (Section 3.3), which allows testing the application without an real device and the Android SDK, the only tool without an alternative for Android applications.

For all these tools, general requirements can be elicited. All tools should be integrated in the IDE or Android SDK if possible, so that they are easy to use. Furthermore all tools should be known by the known by the teaching staff, so that they can support the course participants in best possible way.

Furthermore this chapter shows how to set up the IDE (Section 3.4.1) and the chosen emulator (Section 3.4.2). Anymore it describes how to connect an Android device (Section 3.4.3).

3.1 Development Environment

To develop the application in the practical part of this course, a development environment, the Android-SDK and the Java Development Kit (JDK) is needed. The JDK muss be available at least in version 6 [Inc13h]. The Android SDK can be downloaded from the Android developer website [Inc13h].

For the use in a mobile computing course, the development environment should be known by the participants, easy to use and provide the whole functionality of the Android-SDK. To develop with the SDK, Google offers two alternative bundles to get started with Android development. Each bundle includes an IDE, the Android-SDK and the ADT-Plugin. Both bundles provide everything that is needed to get started with Android development. The IDE shall be easy to use and support the development process. It shall provide a stable environment so that the development process is not interfered by the IDE.

ADT-Bundle The ADT-Bundle (Android Developer Tools) includes the Android-SDK, Eclipse as IDE and the ADT-Plugin. The ADT-Plugin extends Eclipse with some tools

to make the development easier [Inc13b]:

- GUI access to many command line SDK tools
- SDK Graphical Layout Editor
- integrated documentation for Android framework APIs

The IDE Eclipse is widespread and often used for Java development. It works very well together with the Android SDK and is extensible by many plugins. It runs stable and supports the developer while the developing process.

Android Studio This bundle includes the IntelliJ IDE instead of Eclipse. Like Eclipse, the ADT-Plugin is integrated by IntelliJ to provide easy access to the SDK tools. The goal of this bundle is to make Android development easier and faster. But, up to now, Android Studio is only available as early access preview [Inc13i].

Its biggest advantage is that Google tries to make this IDE more intelligent by supporting the developer in many different ways. Its biggest disadvantage is already mentioned above. Android Studio is still in development and only a beta version is available that is not a stable version.

Decision The course needs a stable and reliable development environment. Because Android Studio is still in development, and a beta version is the only available version, the ADT-Bundle (Section 3.1) is used to develop the Android Application. Furthermore, Eclipse is very popular with Java development so that the chance is high that the participants have some experience with it.

3.2 GUI Builder

Developing a mobile application also means to create a usable user interface. To reach this goal it is necessary to test some ideas and create increments. To make the process of creating GUI increments easier, some tools stand to disposal.

The GUI-Builder should provide an easy and fast way to build graphical user interfaces. The user must be able to create, delete and change elements. The user should be able to do the most parts of the GUI development with a What-You-See-Is-What-You-Get editor. The outcome should be available in the current Android project, so that work have not be done twice.

Droid Draw DroidDraw [Dro13] is a *WYSIWYG* user interface designer. It generates XML-Code for each user interfaces that can be used for the XML-Files in the Android project. DroidDraw is easy to get started and easy to use. It is limited to define layouts and generates XML code which can be used in an existing Android project.

MIT App Inventor The MIT App Inventor [Tec13] is a program which allows developing Android applications with a *WYSIWYG* editor. It provides a Designer to build User Interfaces and a “Block Editor” to specify the behavior.

Like DroidDraw, the MIT App Inventor is easy to use. Benefits are also that no coding is necessary to create layout and behavior. Furthermore, it includes its own emulator to test the application. One of the disadvantages is that no Java code can be exported to use it in an existing Android project. Another disadvantage is that the MIT App Inventor is just available as web app so that an internet connection is required.

SDK Graphical Layout Editor The Graphical Layout Editor [Inc13b] from the Android SDK allows creating User Interfaces with a graphical editor or by writing XML-Code. Both the XML-Editor and the graphical editor support many features for rapid UI development.

The integrated layout editor is like the two alternatives above easy to use and easy to get started with. Furthermore, it directly integrated into Eclipse so that no further software is required.

Decision All three tools provides easy ways to develop the GUI. To develop the GUI for this application, the SDK Graphical Layout Editor from the Android SDK is used. It provides at least as much functionality as DroidDraw and the MIT App Inventor Designer. Furthermore, it is integrated into Eclipse, so that there is no need for other standalone software.

The MIT App Inventor is also intended for a somewhat different audience. While the Mobile Computing Course is aimed at people with programming experience, the MIT App Inventor also is aimed at people without this experience.

3.3 Emulator

An emulator allows the developer to test his application without using a real Android device. It imitates most of the functionality of a physical Android device, except for the GPS module. An emulator for this mobile course should be fast and easy to use. The participants needs quick access to test their application. Furthermore, a integration into the development environment is needed, so that the emulator is easy to start.

Android Emulator The Android Emulator is an emulator included in the Android-SDK [Inc13c]. To create an Android Emulator, a Android Virtual Device (AVD) is required which must be created with the AVD Manager [Inc13e]. The biggest advantages of this emulator is, that it is integrated in the Android SDK. It is easy and fast to create a new virtual Device. The authors experience with this emulator is, that it works very slow.

Android-x86 Emulator The Android x86 Project provides Android images for x86 machines [Ax13]. This images can be used to run Android in virtualization software like VirtualBox [Ora13b]. The installation and start of this emulator is easy and fast. To test the developed application, VirtualBox can be connected to Eclipse so that the application can be deployed directly in the virtual machine. This emulator needs also a lot of configuration, which is not always easy. It starts faster than the Android Emulator but it executes applications not always faster.

Genymotion Genymotion is an easy to use and fast emulator for Android [Gen13]. Genymotion provides several emulators that can be run with VirtualBox [Ora13b]. These emulators are easy to integrate in the development process and run very fast. To get the images of the emulators a registration on the Genymotion Website it necessary. After that the emulators are free to use. Genymotion emulators provides Wifi, 3G, Bluetooth and GPS functionality. The emulator boots up very fast and is runs all applications in realtime. Configuration is not necessary.

Decision Although the teaching staff at the Nelson Mandela Metropolitan University has experience with the Android-x86 emulator, the author recommends the use of Genymotion. It is the fastest and easiest to use emulator within this three. It is much faster than the Android-x86 emulator or the integrated emulator among these three.

3.4 Tool Set Up

This section describes how to set up the tools, which were chosen in Sections above. The first tool is the ADT-Bundle from Section 3.1, which has a very easy set up (Section 3.4.1). The GUI Builder 3.2 do not need an extra set up because it is integrated in the SDK. When the ADT-Bundle is installed, the GUI Builder is also ready to use. Section 3.4.2 describes how to set up the Emulator from Section 3.3. The last section (3.4.3) of this chapter shows how to connect a real Android device.

3.4.1 ADT-Bundle

The first step is Downloading the ADT-Bundle [Inc13h] that includes everything that is needed to get started. Latest Version: <http://developer.android.com/sdk/index.html>

After Downloading the zip-file must be unpacked into the installation directory. Open the unpacked directory, open the subdirectory *eclipse/* and launch *eclipse*. After starting Eclipse, the Android-SDK must be updated by starting the SDK-Manager. On Windows this can be done by running the *SDK Manager.exe* file. On Mac OS and Linux is a executable file *android* available in the subfolder *tools/*. To use the Google API it is necessary to download all Google API packages that should be supported. In this case all APIs are downloaded.

3.4.2 Emulator

Visit the website <http://www.genymotion.com/> and sign up for Genymotion. After the registration Genymotion can be downloaded and installed on the system.

Add a new emulator by clicking on the "Add" button. Choose a device from the list and click one more time on "Add" to start the download. To run the emulator VirtualBox must run in the background because Genymotion uses the VirtualBox Engine. Start the emulator with a click on "Play". A new screen with the emulator pops up. The emulator is ready to use. Figure 1 shows the start screen of Geanymotion with the "Add" and "Play" buttons.

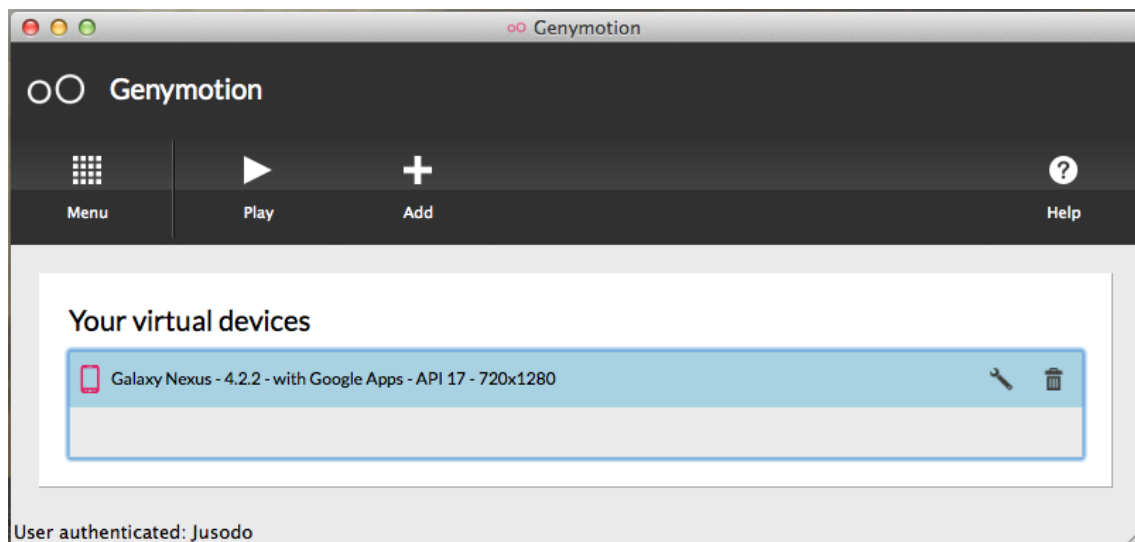


Figure 1: Geanymotion

To check if the device is detected by the ADB-Plugin start a Terminal/ Command line and type the following to get a list of connected devices:

```
1 adb devices
```

If the device is not listed, connect it with the connect command and the IP-address. The IP-address can be found on the emulator by opening the "Genymotion Configuration" application which is on the Android Homescreen. The command to connect the device (with IP-address example):

```
1 adb connect 192.168.56.103
```

3.4.3 Connecting an Android Device

When developing an Android application, an Android device is necessary for testing, because not all device functions can be simulated in an emulator, like the gps module or the accelerometer. To set up an Android device, follow these steps [Inc13m].

- Enable *USB debugging* on the device
 - Android 3.2 or older** Go to *Settings* → *Applications* → *Development*
 - Android 4.0 and newer** Go To *Settings* → *Developer options*
- Connect the Android device with the computer
 - If the operating system is Windows, install the OEM USB Drivers. More information about how to install a driver on Windows: <http://developer.android.com/tools/extras/oem-usb.html>
 - If the operating system is OS X, skip this step.
 - If the operating system is Ubuntu Linux a *udev* rules file is needed. More information about how to define this file: <http://developer.android.com/tools/device.html>

4 Application Requirements

Developing a software is based on requirements, which define the properties and the functionality of the software. Requirements ensures, that the all stakeholders got the same conception of the software.

For this course it is important that all course stakeholders, like the course participants and the teaching staff, have the same idea of the application. Therefore, this chapter provides conceptual information about the application. From a vision document, which gives a rough idea of the application, up to specific requirements, which described more detailed the functionality of this application.

The first section of this chapter is an extended vision of the Android application. It describes the applications functionality, gives a short overview of the development process and, mentions the used tools (Section 4.1). The next section introduces an application which covers a lot of the functionality mentioned in the vision (Section 4.2). Looking at this application can give a rough idea about what the application, which is developed in this course, will be. Afterwards, use cases are identified in two steps. First step, possible use cases are prepared, second step, these use cases are combined to new one. The result is an abstract use case diagram, which describes the main uses cases (Section 4.3). On this basis, user stories are created to provide more details. From each user stories functional requirements and non-functional requirements are derived (section 4.4).

4.1 Application Vision

This section describes the vision of the application. The vision is a guide, which gives a rough idea of the functionality of the application and it creates a common context between the course participants, so that everybody has an idea, what the application is about. As mentioned in Section 1.1 this course takes place at the Nelson Mandela Metropolitan University in Port Elizabeth, South Africa. Near the university is the Kragga Kamma Game Park which is used to give the application a local context. This context is chosen to cover the described learning outcomes from Section 2.1.

The Kragga Kamma National Park offers its visitors the chance to see a lot of wild African animals in their natural environment. All animals can move freely in this territory. Visitors get the opportunity to drive through this Game Park with their own car or to join a two hour guided tour [Par13]. At the entry of the Game Park, a board with the last animal sightings is lined up. This board is updated once a day and the sighting description is not very precise. Because the animals can move freely in the park, the sighting information are quickly out of date. To provide actual information to all visitors, a new approach is

needed. The idea for this Android application is, that a visitor who sees an animal, can make these information available to all other visitors of the Kragga Kamma Game Park. Other park visitors with Android devices can download this information. In order to do that, the visitor uses his Android device to localize himself and upload information to a server. This information are the kind of animal and the GPS data and optional a photo. To make this easy, the visitor uses an Android application on his mobile device. The target group for this application are users with Android devices, who visit the Kragga Kamma Game Park. The Android device should have a GPS sensor and must be able to connect to the internet.

The application allows the user to do two main things: add a new sighting to share it with other visitors, and view sightings that other visitors have shared. To share the information about the whereabouts of an animal, the user must share the information about his actual abode, the actual time and the animal he has seen. On an extra view he can choose the animal, localized himself and add a photo. To be localized, GPS must be enabled or the possibility to be localized via wireless networks. To add a photo the user has two options. He can take a new photo with the devices camera or he can choose a photo from the gallery. When the user takes a new photo for a sighting, the GPS data should be stored. The GPS data allows to see on a map where the photo and the sighting was taken. The application must set a tag that contains the information that the photo was made from this application.

If it is possible to localize the user and he chooses the animal he has seen, all information are stored to an database which stores this information. To upload the information an internet connection is necessary. If the connection is not enabled, the application must point out that the internet connection is required to continue. To upload the information to an server in order to share it with other visitors, the user must synchronize the application with the server. A user who wants to get informed about the last animal sightings can download the sightings information and pictures from the database. The application connects automatically with the database, if the users orders new sighting information. If the connection is not available the application informs the user, that the download is not possible at the moment. In Order to converse the battery and the data volume, the download and the upload of new information must be triggered by the user.

The application provide two ways to view the sightings. On way is to display a map which shows all sightings to the user. This map shows with their default setting the Kragga Kamma Game Park. To show the map, an internet connection is required. The second way provides an alternative view to display all sightings. On the map the user can view his current position. To do this, the GPS or the localization via wireless networks

must be enabled. A small symbol indicates the position of the user. Furthermore the map displays the last animal sightings. Small markers suggest the points, where animals were seen. The user has some options to filter the sightings, to see only one specific animal. This filter should be available from the sightings overview as well as from the map. The sightings overview should provide the possibility to sort the sightings by attributes like name, distance to the user and time.

The user gets also the opportunity to look at more detailed information about the sightings information that are stored on his device. This information are a picture of the animal and the time at which the photo was taken. The displayed picture can be standard picture, which is stored on the device or the picture he downloaded from the database with the sighting information. Furthermore the user can share a sighting via external services like Facebook or Flickr if he views the detailed information.

The external database stores information from the animal sightings which are the kind of animal, the GPS information and the time-stamp. This information were uploaded when a user added a new animal sighting and this information were also provided to other user who wants to inform about new animal sightings.

To develop the mobile application described in the text above, an iterative approach is chosen. Each iteration covers a new topic and the outcome of each day will be a new increment of this mobile application. The topics and outcomes are (see also [1.1](#) and [2.1](#)):

Iteration 1 Introduction to the development environment and the Android framework. At the end of this iteration, all steps from setting up an Android project to the point of deploying the Android application have been taught and the students are able to do these steps. Furthermore the Android project, which will be worked on the next iterations, is set up.

Iteration 2 Designing and building a graphical user interface. As outcome of this iteration the increment has a basic GUI, which will be extended in the following days.

Iteration 3 Using databases and sensors. The increment of the third iteration includes the usage of the camera, the GPS and the database. It is possible to take photos, get the GPS data and to store data about animal sightings in a database on the device.

Iteration 4 Integration of external services. As outcome of this iteration, the increment provides the functionality of sharing photos with external services like Twitter, Flickr or

Facebook. It uses a map service like Google Maps and it is possible to send and to receive sighting information from an external database.

Iteration 5 The fifth iteration is for finish presentations of the course participants and for further presentation from external partner.

The tools that are used for this development process are the Android SDK, Eclipse as IDE 3.1, the SDK graphical layout editor 3.2 as GUI builder and the Genymotion emulator 3.3 for testing the application (see also Section 3 regarding the tool environment).

4.2 Similar Application Africa Life

An application that provides very similar functionality to what is described in section 4.2, is the application *Africa: Live*, which is available for Android [Tra13a] and iOS [Tra13b]. There is also a web application to the newest animal sightings [Tra13c]. This application is reviewed, to show how this scenario can be implemented. Furthermore it can inspire to choose another way of implementation.

Africa: Live is published by Satpack Travel, is currently available in version 3.1 at the Google Play Store and requires Android version 2.2 or higher. The application allows the user to see, which animal was sighted where in Africa. The sightings are shown as pinheads on a map. The user can make a long click on the map to add a new sighting or pick one of the pinheads to get more information about that specific sighting. Adding a new sighting means to enter a long list of information like the name, the group, the species, the visual quality of the sighting and much more. A photo can also be added to the sighting by taking a new one with the camera or choosing one from the gallery. The application includes Facebook as an external service to publish comments on their Facebook page.

Overall the application *Africa: Live* does all the tasks that are described in the application vision (see section above 4.1). For example, one thing that can be improved, is the viewing of sightings. When there are a lot of sightings in one place, it can be very hard to pick a specific pinhead one on a map. The user has to zoom a lot, which means that the map has to update constantly to provide a more detailed view. This problem can be solved by using an alternative way to provide an overview about the sightings, like a list or a grid view. Furthermore this application provides some content that is not covered by the course application. Like offline Maps for some Game Parks (but only in a paid version), a field guide with information about different animals, a social media stream (mostly in the Web App) and some general safari tips.

Africa: Live is a good example how this kind of application can work. It shows a possible usage concept and shows how task can be done in good or a less good way and inspired to create new concepts.

4.3 Use Cases

This section describes the main functionality of the application in the form of use cases. Use cases are necessary to describe the main functionality of the application and which stakeholders are involved. To identify the main use cases and their stakeholder, two steps are necessary. At the first step, a detailed use case diagram is created, this helps to identify stakeholder and actors. Furthermore it gives an overview about the use cases, which represents the functionality of the system. The second step is to combine these use cases as far as possible, to find the main use cases for this application. This approach is described by Edward Yourdon [You90]. This use cases will be specified in the next section User Stories 4.4.

For each stakeholder and actor, use cases are identified. Figure 2 shows these use cases in a detailed level. It shows the relevant stakeholder and all involved actors. Since the user is the main and only stakeholder the most use cases affects him. The use cases can be traced back to the application vision in Section 4.1. The described scenario is pictured here in use cases, to focus on the important functionality. After that, this use cases were combined to new use cases, which represent in a abstract way the main functionality of this application. Figure 3 shows this main use cases. This diagram contains only the four main use cases.

All use cases, that have to do with adding a new sighting are combined in *add a sighting*. This implies choosing an existing photo from the gallery, taking a new one with the camera and getting all other required information. In *view all sightings* are all use cases summarized, that are needed for showing and providing all available sightings. The use case *view map* implies all use cases, that affects viewing a map, view a sighting or get the own current position. The GPS and the map service are the involved components, that interacts with the application. The last use case *share a sighting* includes all necessary use cases, that affects the external services and the user to provide the functionality of sharing a photo.



Figure 2: Detailed use case diagram

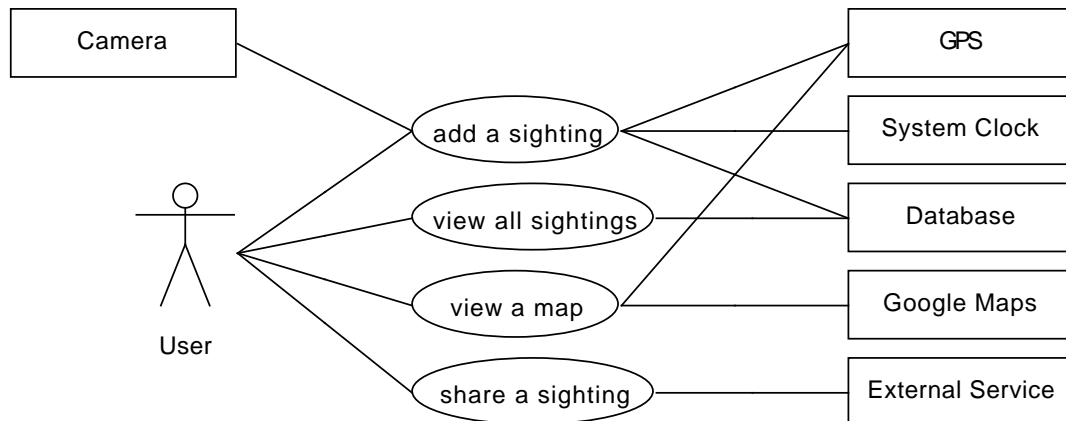


Figure 3: Overview of the main use cases

4.4 User Stories

The following sections describe the use cases from Figure 3 in more detail in form of user stories. Each use case from Figure 3 represents a user story and provides information about the name, the actors, the components, a description, an abstract basic flow, an optional precondition, a trigger and the iteration that are affected by the user story. All information, that are necessary to create this user stories, are provided by the application Vision in Section 4.1 and the detailed use case diagram Figure 2. Stephan Kleuker describes in his book “Grundkurs Software Engineering mit UML” (page 62 ff.) how to record use cases in this form [Kle11].

Afterwards, for each user story requirements are elicited and listed. The requirements are elicited based on the user stories, so that each requirement can be assigned to a user story. Each requirement has a unique number to identify and reference it. This enables traceability of the requirement and helps to evaluate the outcome application. The requirements can also be used to create tasks for the course participants

To have an consistently formulation for the requirements, the words ‘shall’ and ‘should’ are used to describe different levels for the requirements. This two words are used in the context defined by S. Bradner from the Network Working Group [Bra97]:

Shall Is equivalent to ‘must’ and means, that it is an absolute requirement.

Should Is equivalent to 'recommended' and means, that there is a reason why this requirement is needed and a good argument is needed to ignore the requirement.

In the following descriptions and listing the application is called *WildlifeAfrica*.

4.4.1 Add a Sighting

Table 1: User Story: Add a Sighting

Name	Add a sighting
Actor	User
Component	Database, Camera, GPS, System Clock
Description	The user wants to add a new animal sighting to the external database.
Basic Flow	<ol style="list-style-type: none"> 1. Choose the animal's name. 2. Take a photo or choose one from the gallery. 3. Get the current position (location data). 4. Get the current time. 5. Write a description. 6. Add the observer's name. 7. Store sighting information.
Precondition	GPS and internet connection are enabled.
Trigger	The user wants to add an animal sighting.
Iteration	GUI (2.1), Databases and Sensors (2.1) and externals Services (2.1)

Requirements

- 1.01** The SQLite database shall store gps information.
- 1.02** The SQLite database shall store time data.
- 1.03** The SQLite database shall store the sighting description.
- 1.04** The SQLite database shall store the observer's name.
- 1.05** The SQLite database shall not store photos.
- 1.06** WildlifeAfrica shall store photos on the device.
- 1.07** WildlifeAfrica shall be able to add new entries to the SQLite database.
- 1.08** WildlifeAfrica shall be able to delete existing entries in the SQLite database.

- 1.09** WildlifeAfrica shall be able to read entries from the SQLite database.
- 1.10** WildlifeAfrica shall be able to upload sighting information to a MySQL database.
- 1.11** WildlifeAfrica shall be able to download sighting information from a MySQL database.
- 1.12** WildlifeAfrica shall be able to access the camera.
- 1.13** WildlifeAfrica shall be able to access the gallery.
- 1.14** WildlifeAfrica shall be able to access the device storage.
- 1.15** WildlifeAfrica shall be able to get location data.
- 1.16** WildlifeAfrica shall have access to the system clock.
- 1.17** WildlifeAfrica shall have a view to add a new animal sighting.
- 1.18** The add-sighting-view shall have a choice box to choose an animal.
- 1.19** The add-sighting-view shall get the GPS data automatically.
- 1.20** The add-sighting-view shall have the possibility to save the sighting to the database.
- 1.21** The add-sighting-view shall have the possibility to abort the act of adding a new sighting.
- 1.22** The add-sighting-view should have the possibility to navigate back to the previous screen.
- 1.23** The add-sighting-view shall have a text field to enter a description.
- 1.24** The add-sighting-view shall have a text field to enter the observer name.
- 1.25** The add-sighting-view shall have the possibility to access the camera.
- 1.26** The add-sighting-view shall have the possibility to access the gallery.

4.4.2 View all Sightings

Table 2: User Story: View all sightings

Name	View all sightings
Actor	User
Component	
Description	The user wants to look at all sightings, that are stored on his device. The user wants to see a specific sighting or he wants to see the details of a specific sighting. To find a specific sighting, the user can filter or sort all available sightings. Furthermore he wants to synchronize sightings with the database.
Basic Flow	<ol style="list-style-type: none"> 1. Synchronize with the database. 2. Browse through all sightings. 3. Select a sighting to view details.
Precondition	Sightings must be stored on the device.
Trigger	The user wants to view all or a specific sighting.
Iteration	GUI (2.1), Databases and Sensors (2.1), external Services (2.1)

Requirements

- 2.01** WildlifeAfrica shall display all sightings that are stored on the device.
- 2.02** WildlifeAfrica shall have a view to display details about a specific sighting.
- 2.03** WildlifeAfrica shall be able to filter the available sightings.
- 2.04** The sighting-details-view displays the animal's name, a photo, the time stamp, the description and the observer's name.
- 2.05** The sighting-details-view shall have the possibility to navigate back to the previous screen.
- 2.06** The sighting-details-view shall have the possibility to add a new sighting.
- 2.07** The sighting-details-view shall have the possibility to share a sighting via external service.
- 2.08** The all-sightings-view should be able to sort the sightings by name (a to z and z to a).
- 2.09** The all-sightings-view should be able to sort the sightings by distance.

- 2.10** The all-sightings-view should be able to sort the sightings by time.
- 2.11** The all-sightings-view shall have the possibility to navigate to the map.
- 2.12** The all-sightings-view shall have the possibility to add a new sighting.
- 2.13** The all-sightings-view shall have the possibility to synchronize with the database.

4.4.3 View map

Table 3: User Story: View map

Name	View map
Actor	User
Component	Google Maps, GPS
Description	The application displays a map with the last animal sightings and the location of the user. So the user can see where an interesting animal were sighted. By default the application shows a map of the Kragga Kamma Game Park. The user is able to filter for a specific animal.
Basic Flow	<ol style="list-style-type: none"> 1. View the map. 2. See the own position on the map. 3. Filter sightings to see only a few. 4. Tap on a marker to display further information. 5. Tap on the marker information to view sighting details.
Precondition	Internet connection is enabled. GPS is enabled. Sightings are stored on the device.
Trigger	The user wants to see his position on a map and/ or he wants to see animal sightings on a map.
Iteration	GUI (2.1) and External Services (2.1)

Requirements

- 3.01** WildlifeAfrica shall have access to the GPS module.
- 3.02** WildlifeAfrica shall have access to Google Maps.
- 3.03** WildlifeAfrica shall have a view to display the map.
- 3.04** WildlifeAfrica shall be able to navigate to the map-view.
- 3.05** The map-view shall be able to display the user on the map.

- 3.06** The map-view shall have a marker to display an animal sightings on a map.
- 3.07** The map-view shall have a button to show the users location on the map.
- 3.08** The map-view shall have the possibility to navigate to the view with all sightings.
- 3.09** WildlifeAfrica shall be able to filter the sighting information.
- 3.10** The starting point for the map-view should be the Kragga Kamma National Park.
- 3.11** The map-view shall have the possibility to add a new sighting.
- 3.12** The map-view should be able to synchronize the sightings with the database.
- 3.13** The map-view should be able to get location information.

4.4.4 Share a Sighting

Table 4: User Story: Share a sighting

Name	Share a sighting
Actor	User
Component	External Service
Description	The user wants to share a sighting via an external service. The photo and further information about a specific sighting are uploaded to an external service.
Basic Flow	<ol style="list-style-type: none"> 1. Choose a sighting. 2. Choose service to share. 3. Share the sighting.
Precondition	The user must have a sighting on his device. Internet connection must be enabled. User must be logged on the service.
Trigger	The user wants to share a sighting with his friends and family.
Iteration	GUI (2.1) and external Services (2.1)

Requirements

- 4.01** WildlifeAfrica shall be able to share information with an external service.
- 4.02** WildlifeAfrica shall be able to notify an external service.
- 4.03** WildlifeAfrica shall be able to share a photo with an external service.

5 Architecture

This mobile computing course is divided in several iterations and the application, that will be developed, will be expanded in each of this iterations. Therefore it is necessary that the application provides an architecture, that can be easily expanded in each iteration. The software architecture for this application needs to be simple and easy to understand, because in an intensive course, there is not much time to think about a complex architecture. Furthermore the architecture requires a minimum dimension of modularity to ensure, that this application can be developed in increments.

This chapter describes the reference architecture, which follows the Model-View-Presenter pattern (Section 5.1) and the final architecture for this application (Section 5.2).

5.1 Model-View-Presenter

The basic architecture for this application is the Model-View-Presenter pattern. This pattern separates the model and the view completely from each other, which causes a better separation of the components. The view and the model are connected through a presenter [Fow13].

The Model-View-Presenter (MVP) pattern is emerged from the Model-View-Controller pattern [Fow13]. In the MVP pattern, figure 4, the view is separated from the presenter and does not contain any behavior to react to user input. This component just structured the presentation of the data. All user input is handled by the presenter, which knows how to react. The presenter is responsible to update the model and the view. It prepares the data to represent them in a view or to store them in the model. The model represents all necessary data.

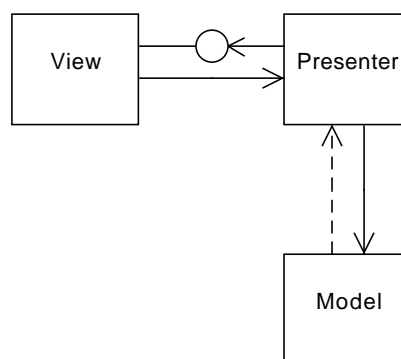


Figure 4: Model-View-Presenter pattern

5.2 WildlifeAfrica Architecture

Developing for Android means to work with the Android SDK, which contains different components to build an Android application (see Section 6.5 for the components). If these components are used, they already form a structure, that matches well to the MVP pattern. The view components are represented by XML files which do not contain any logic. Activities works as presenter, handling the user input, communicates with the model and updates the view. And the model contains data structures, database access and functionality encapsulated in modules. Figure 5 shows how all these components inside the Model-View-Presenter pattern. All these components will exist at the end of the last iteration.

View The view is defined and represented by XML layout files. This XML files are used by activities and other objects that are displayed inside an activity. They do not contain any logic and all user input is forwarded to the activities.

Presenter This contains the activities, which are the center points of Android applications, because they handle the user input, switches between the screens, interact with the services and update the view. Adapters are used by activities to update lists, which are presented to the user ,so he can interact with them. Comparators also are used by activities to sort lists by different criteria.

Model The model contains different services, which are separated by their functionality. The *LocationService* provides location data and the *SyncService* connects with an server to synchronize the local database with a database on a server. The *DataTransferObject* is used to transport data from the *Database* to the presenter, and backwards. The *Database* encapsulated the internal SQLite database, to provide access to the database.

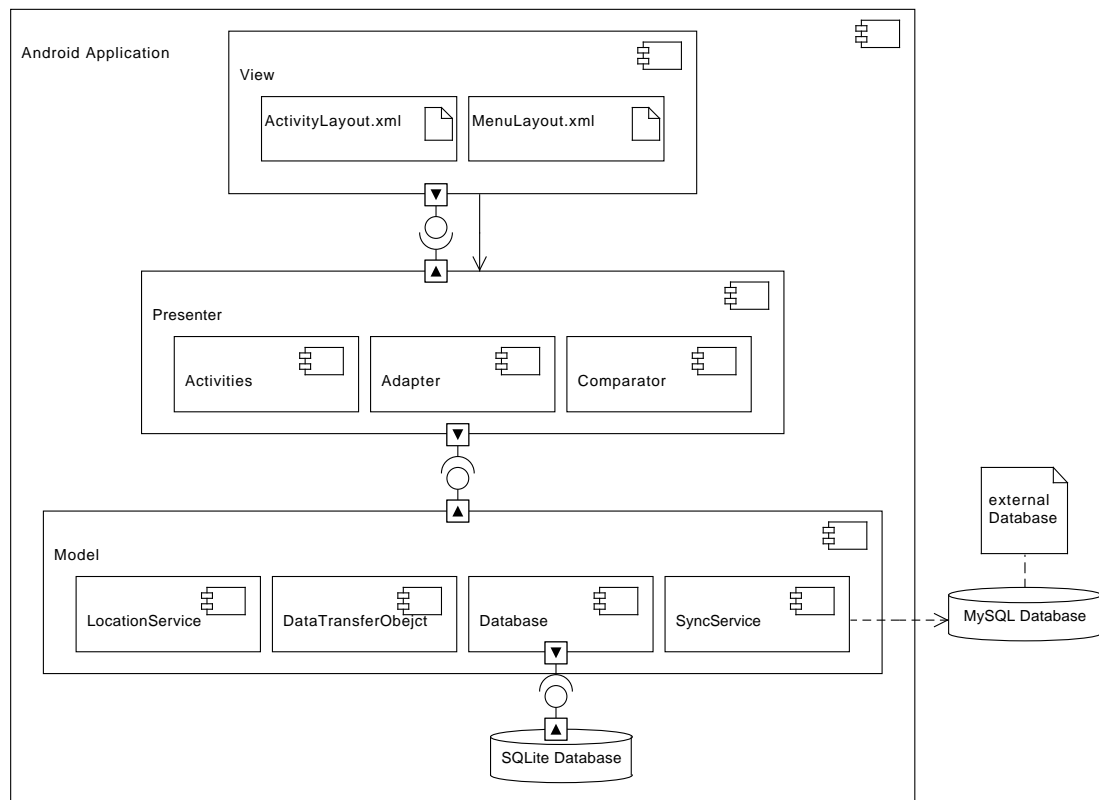


Figure 5: Application screen structure

6 Iteration 1: Android Framework

The first iteration gives a short introduction into the Android Framework, which forms a basis for the following iterations. This chapter includes a short “Hello World” tutorial, Section 6.1, to match the learning outcomes described in Section 2.1. Afterwards the structure of an Android project is presented in Section 6.2. The following sections introduces Android specific elements (Section 6.3, 6.4), components (Section 6.5) and the SQLite database (Section 6.6), that are used in the following iterations. The next section shows, how to set up the Android project for the WildlifeAfrica application (Section 6.7), which will be used in the further iterations. The last section is about the time it took the author, to go through this iteration (Section 6.8).

6.1 Hello World Application

This “Hello World” application shows how to set up a new Android project and how to deploy an application on an Android device. Especially the deploying is relevant for the following iterations, because deploying the application on a device or emulator is very helpful to test the application and will be done a lot of times in the development process.

To create a new Android project click in Eclipse to *File* → *New* → *Android Application Project*. Eclipse starts a wizard to create a new Android project. Enter an *Application Name*, a *Project Name* and a *Package Name*. The *Application Name* can be for example “HelloWorld” or any other name because this project is not used after deploying any more. Click on the *Next* button and follow the wizard until it is complete.

The created project is a runnable application, that will present the words “Hello world!” on the screen. To deploy this application on a real device or on an emulator, do a right click on the project folder in the Eclipse “Package Explorer”. Choose *Run As* → *Android Application*. Eclipse searches for available devices (no matter if real device or emulator) and starts a wizard to choose a connected device from a list. Choose a device and click on the *OK* button to deploy the application on the device.

6.2 Project Structure

The project structure is created by the project wizard and should not be modified. By default this structure contains everything that is required for a runnable application. Figure 6 shows the structure in the Eclipse *Package Explorer* view. The two important directories are short described in the following paragraphs.

HelloWorld\src This is the directory for the java files.

HelloWorld\res This directory contains resources like images or XML files. It contains several subdirectories, the relevant directories are described in Section 6.4.

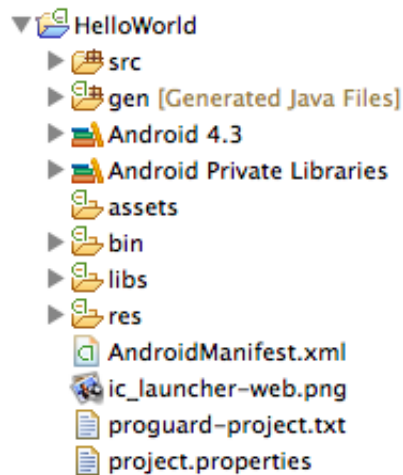


Figure 6: Android project structure

6.3 Manifest File

Each Android application contains its own `AndroidManifest.xml` file in its root directory, see figure 6. This file defines the components and properties of the application, for example requirements to the Android platform or the required permissions to access the camera or the internal storage.

6.4 Resources

Resources are all kind of data, that are used by the application, but is not a real part of it, like text or images. All this resource files are stored int the directory `res/`. Resources are defined in XML files, except images, audio and video files. The framework provides automatically access to all resources via the `R` class, which allows to reference a resource in the source code. For example `R.layout.activity_sighting_list` reference to a XML layout file in the directory `res/layout` [Sta12], [Mei12].

res/layout This is the most important resource for an Android application. This directory stores all XML files, that defines the layout of a screen or a part of a screen.

res/menu This directory stores layout files for the menu bar of an application.

res/value Contains resources for *String*, *Dimenstions* and *Styles* also in XML files.

res/drawable is for image files that are used in the application.

6.5 Components

The Android framework provides four main components, which are the main blocks to build an Android Application. Each component has its own operation area. The four main components are *Activities*, *Services*, *Broadcast Receiver* and *Content Provider*. The three first components are used to build this application for the mobile computing course. They are shortly introduced in the following paragraphs. This tree components require special message to be activated. This message is called *Intent* and also introduced in the following paragraphs [Inc13d], [Sta12].

Activity Activities represent the visible part of an Android application. Each activity provides a single screen and a user interface the user can interact with.

Services A service is a component without an user interface. It runs in the background of the application to perform long or complex tasks.

Broadcast Receiver This component reacts to system-wide events, which were triggered by other components.

Intent An Intent is a message that calls a certain component. Furthermore, it can transport data between components.

6.6 SQLite Database

The application, that will be built in the following iterations, requires a database to store data. The Android framework includes with SQLite a relational database system, which provides a Java interface for usage. The database is created, designed and managed by the Android Framework.

6.7 WildlifeAfrica Project

This section describes how to set up the project for the application WildlifeAfrica and how to integrate the Google Play Service SDK [Inc13j], which are used in iteration 4 to show the map (see Section 9.2). Three steps are necessary to work with the Google Play SDK.

First, download and install the Google Play SDK. Second, set up the SDK as a library project. Third, reference the Google Play SDK to the main project.

6.7.1 Project Setup

Start the project wizard in Eclipse with *File* → *New* → *Android Application Project* and set the *Minimum Required SDK* to “API 11: Android 3.0 (Honeycomb)”. This is necessary to work with the action bar, which is used to navigate through the application, see Section 7.5.1. If the project is already created the *Minimum Required SDK* can be changed in the *AndroidManifest.xml* file inside the *uses-sdk*-tag.

6.7.2 Download Google Play SDK

To download the Google Play SDK open the SDK Manager. In Eclipse select *Window* → *Android SDK Manager*. Scroll down to the bottom of the list, open the entry *Extras*, select the entry *Google Play services* and click on the *Install* button.

To get the map work in an emulator also make sure that the *Google APIs* of API level 17 or higher is installed. To do this, click on the entry *Android 4.2.2 (API 17)* and check if *Google APIs* is installed.

6.7.3 Install Google Play SDK

The next step is to set up the Google Play SDK. In Eclipse click on *File* → *Import* → *Android* → *Existing Android Code into Workspace*. In the wizard click on *Browse...*, browse to the directory where the Android SDK is installed and go to */extras/google/google-play-services/libproject/google-play-services-lib/*. Mark the checkbox with *Copy projects into workspace* and finish the wizard.

6.7.4 Reference Google Play SDK

The last step is to reference this library in the WildlifeAfrica project. Rightclick on the WildlifeAfrica project in the *Package Explorer* and click in the context menu on *Properties*. In the new window click on *Android* on the left side. From the group *Project Build Targets* choose *Google APIs* with a API level of 17 or higher. In the group *Library* click on *Add...* and choose from the list the previous added Google Play project. Click on *Apply* and on *OK* to close the window.

6.8 Time

Setting up the “Hello World” example and deploying it on a device is a easy and fast task, that can be done in about 15 Minutes. Preparing the Android project for the following iterations took about 30 Minutes. This task can took longer, when only a slow internet connection is available and the students have to download the Google Play SDK by themself.

7 Iteration 2: GUI

The goal of this iteration is to create the first increment of the application *WildlifeAfrica*, by describing the implementation. As described in Section 2.1 this increment should provide a graphical user interface and the user should be able to navigate through the application.

In a first step, it is necessary to define, which screens the application should contain and how to navigate between this screens (Section 7.1). Following pen and paper mock-ups are created to design the layout for the screens (Section 7.2). After creating a structure and a design for the application, the implementation starts by editing the Android Manifest (Section 7.3) and creating some resource files (Section 7.4). Afterwards, the activities are created. At first the XML layout files for the activities are created (requirements 1.17, 1.18, 1.20, 1.23 to 1.26, 2.01 to 2.04, 2.07 to 2.10, 2.13, 3.03, 3.12) and afterwards in Section 7.5 the navigation between the screens (requirements 1.21, 1.22, 2.05, 2.06, 2.11, 2.12, 3.04, 3.11). Following the rest of the activities is implemented. Because the implementation of the activities is very similar only one implementation is shown completely (Section 7.7). All other Sections (7.8 to 7.11) show only important aspects of the implementation. Section 7.12 shows the implementation of the class *Sighting*, which is frequently used in this application to transfer data. The last section is about the time it took the author to go through this iteration (Section 7.13).

7.1 Navigation and Screens

This section describes the relation between the use cases based on their dependencies, shown in figure 7. Because not every functionality has to be available on every screen, the use cases are grouped to define the separate screens, each screen conforms to one activity.

As outcome, the use cases *take a photo* and *choose photo from gallery* are aligned behind the use case *add sighting*. This means that the screen for adding a sighting needs the possibility to access the camera and to access the gallery.

Both use cases *view all sightings* and *view sightings on a map* point to *filter sighting* and *view sighting details* because this two options must be available on both screens. The use case *view sightings on a map* points also to *view own position on a map* because this is only possible if the screen shows a map. *View all Sightings* points furthermore to the use case *sort sightings* because a list or grid view can be sorted by different options.

View sighting detail refers to *share a sighting*, so that the sharing can only be made when the sighting details are displayed.

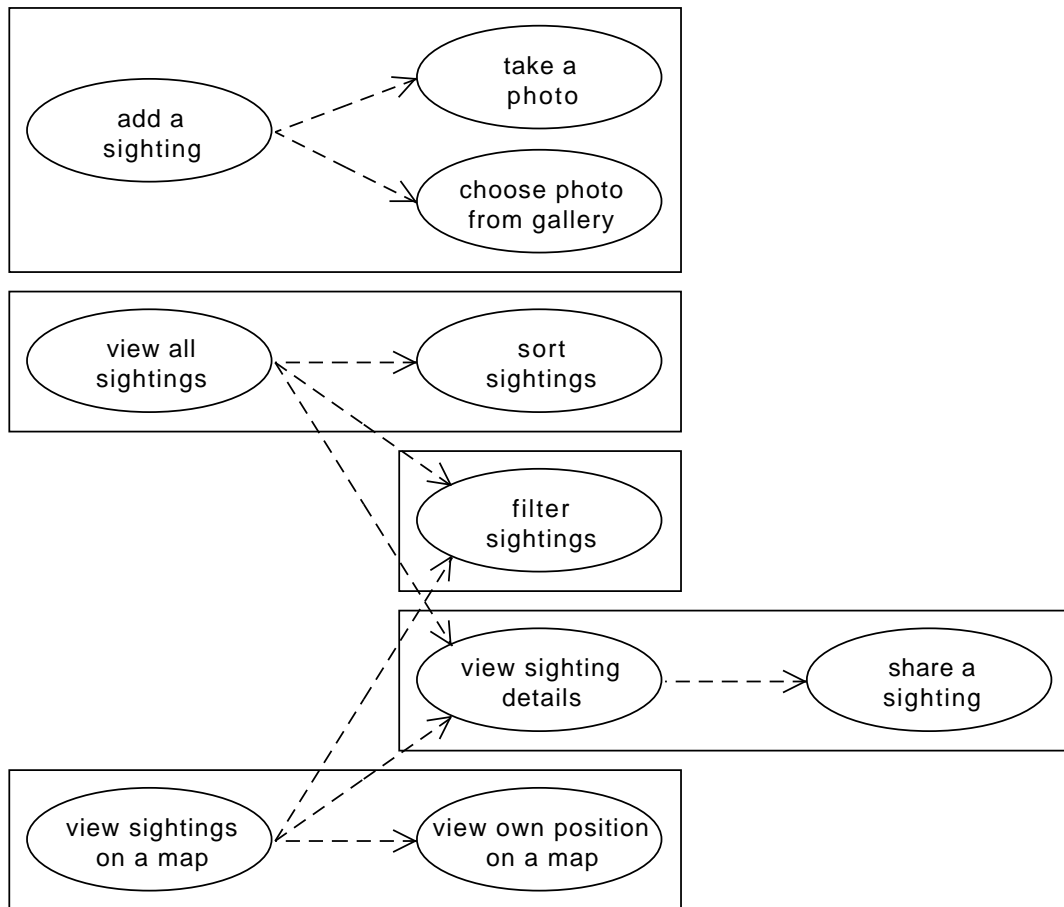


Figure 7: Application screen structure

Add a sighting refers to the user story *add a sighting* (Section 4.4.1). This screens enables the user to add a new sighting to the database. It provides all graphical input fields to collect the required sighting information. It also allows the user to take a photo or to pic up a photo from the gallery. For this two options are none further screen required, because the application starts the Android default camera to take a photo or uses the Android default gallery to select a photo.

View all sightings refers to the user story *view all sightings* (Section 4.4.2). This screens provides an overview over all available sightings. The user can sort the sightings, filter or pic one to get more detailed information.

View sightings on a map refers to the user story *view a map* (Section 4.4.3). This screen shows the user a map and shows the available sightings on the map. Also on this screen the user has the possibility to filter the sightings, filter sightings or to pick out one sighting to get detailed information. Because this screen provides a map, it can also provide the possibility to show the users position on the map.

View sighting details refers to the user story *view all sightings* (Section 4.4.2), *view map* (Section 4.4.3) and *share a sighting* (Section 4.4.4). This screen displays detailed information about a specific sighting. These detailed information are a photo, the animal's name, a timestamp, a description and the name of the user who added the sighting. It also allows the user to share the information via external services.

Filter sightings refers to the user story *view all sightings* (Section 4.4.2) and *view map* (Section 4.4.3). This allows the user to view just specific sightings. This functionality becomes an extra screen because it must be accessible from *View all sightings* and from *View sightings on map*.

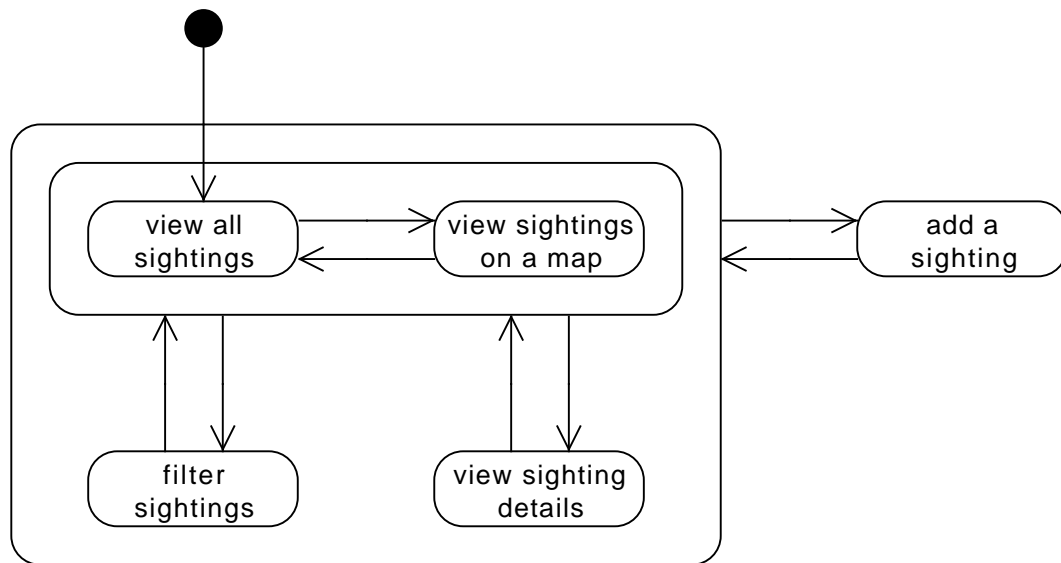


Figure 8: Structure of the screens

Figure 8 shows the required screens and the navigation between the screens. The screen to *add a sighting* can be reached from all other screens, to make sure that the user can

add a sighting at any time. The screens to filter sightings and to view sighting details can be reached from the screen with the overview of all sightings, as well as from the screen with the map.

7.2 Mock-Up

Mock-ups are used to create a first idea of what an application will look like and how it will behave. They simplify the communication about design and behavior by visualizing it. Creating a mock-up can be done with software tools or with pen and paper, which is the faster way. All mock-ups in this section are created with pen and paper. In the context of this course, the mock-ups are used to design the application. The mock-ups are the base for creating the layout files for the activities. They make it easier to work determined on the task.

Figure 9 shows a mock-up for the screens *view all sightings* and *view sightings on a map* from the Section above. Both screens have a very similar list of navigation icons at the top of the screen. A legend for the icons is shown in figure 12 at the end of this section.

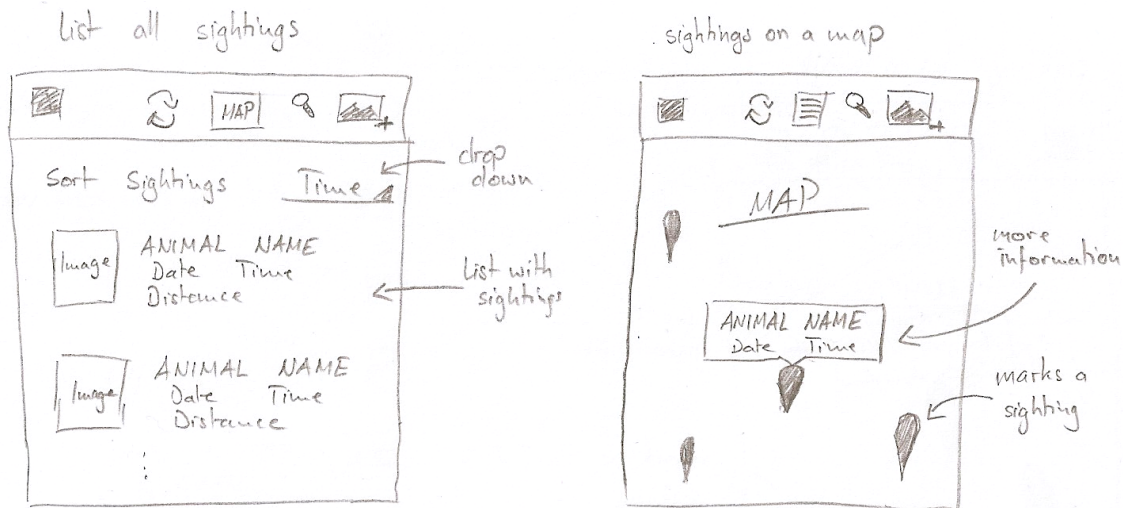


Figure 9: Mock-up: sightings in a list (left) and on a map (right)

Mock-ups for the screens *filter sightings* and *view sighting details* from the Section above are shown in figure 10. The image on the left side shows the screen to filter the sightings and the image on the right side shows the screen to view sighting details.

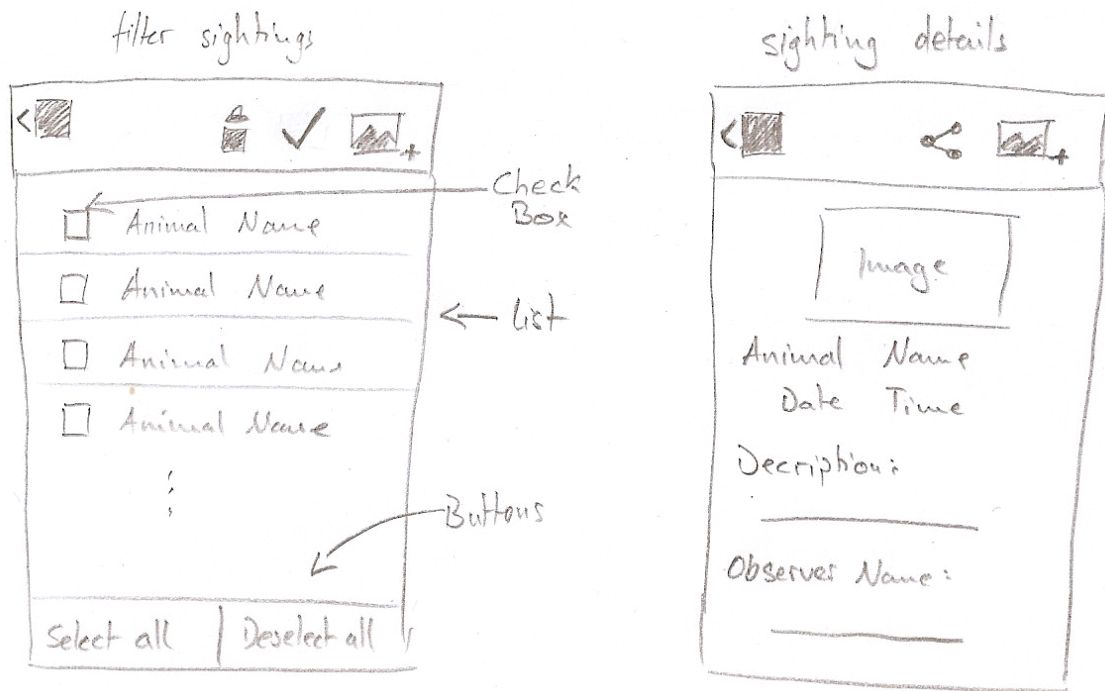


Figure 10: Mock-up: filter sightings (left) and sighting details (right)

Figure 11 shows a mock-up for the screen to add a new sighting and figure 12 shows a legend for the icons that are used the figures 9, 10 and 11

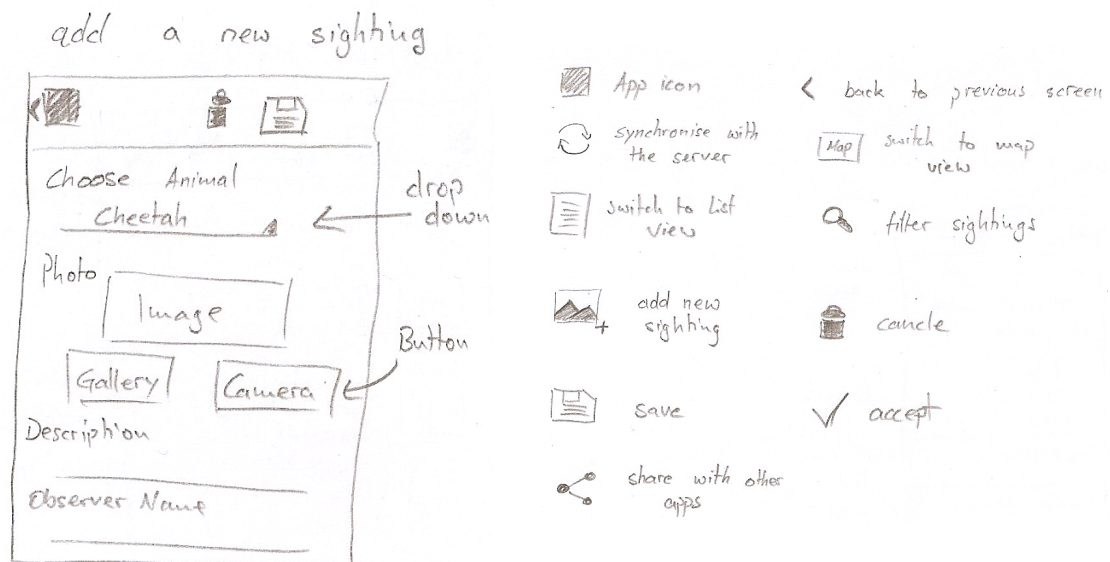


Figure 12: Mock-up: legend for the icons

Figure 11: Mock-up: new sighting

7.3 Android Manifest

All created activities need to be listed in the Android manifest, so that these components can be used. For each screen in figure 8 an activity is created and registered in the Android manifest. The listing 1 below shows how to list the activities in the *application*-tag. The first activity, which is created within the Android-project, is already be listed. To all activities, except the *SightingMapActivity*, the statement *android:screenOrientation="portrait"* is added, to specify, that these activities are used in the portrait mode.

```
1  <activity
2      android:name="uolnmmu.wildlife.presenter.SightingListActivity"
3      android:screenOrientation="portrait" >
4      <intent-filter>
5          <action android:name="android.intent.action.MAIN" />
6          <category android:name="android.intent.category.LAUNCHER" />
7      </intent-filter>
8  </activity>
9  <activity
10     android:name="uolnmmu.wildlife.presenter.AddSightingActivity"
11     android:screenOrientation="portrait" >
12 </activity>
13 <activity
14     android:name="uolnmmu.wildlife.presenter.SightingDetailsActivity"
15     android:screenOrientation="portrait" >
16 </activity>
17 <activity
18     android:name="uolnmmu.wildlife.presenter.SightingFilterActivity"
19     android:screenOrientation="portrait" >
20 </activity>
21 <activity android:name="uolnmmu.wildlife.presenter.SightingMapActivity"
22     >
```

Listing 1: Manifest: register activities

7.4 Resource Files

For this project two text resources are defined, which will be used to display text. A *String-Array*, which contains a list of animal names and another *String-Array*, which contains the text for the filter options. Listing 2 shows exemplary how to define the text resource for the animal names in XML. The second resource is created with the same pattern.

```
1  <string-array name="animalOverview_array">
```

```
2    <item>Blue Duiker</item>
3    <item>Bontebok</item>
4    ...
5    <item>Zebra</item>
6    <item>Other</item>
7  </string-array>
```

Listing 2: Resources: String-Array

7.5 Activities

Each created Activity is a generalization of the class *Activity*. Furthermore all activities uses the action bar (see next Section 7.5.1), to navigate between screens and provide further functionality. Activities are started by intents. All activity classes are stored in the package *uolnmmu.wildlife.presenter*.

7.5.1 Action Bar

The action bar is available for all Android devices with Android 3.0 or higher (API level 11), that requires to set the minimum SDK API level in the Android manifest to 11 [Inc13a]. The layout for the action bar is defined in separate XML layout file, which are stored in the directory: *WildlifeAfrica/res/menu*. The creation and implementation for the several action bars, follows every time the same pattern. Following the creation and implementation of the action bar from the *SightingListActivity* is exemplary shown. The figure 13 shows the outcome.

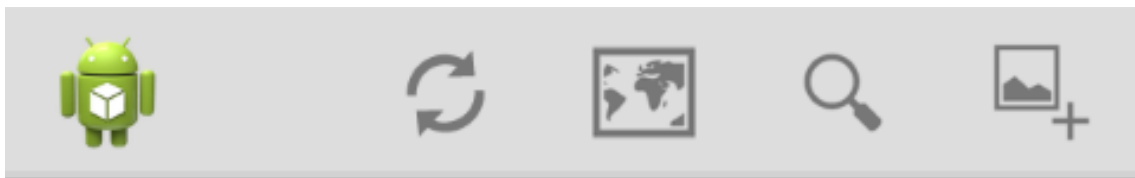


Figure 13: Action Bar for the SightingListActivity

Layout

The layout file *menu_sighting_list.xml*, shown in listing 3, contains four items. Each item represents one icon in the action bar. Each item has the same attributes with different values.

android:id	gives this item an id to be identified
android:icon	allows to set an icon. This icons are stored in the directory <i>WildlifeAfrica/res/drawable-xhdpi</i>
android:title	is the title that will be shown if no icon is available. There are two ways to set a title. The first is shown in the listing below. A String-Resource is referenced, which are stored in <i>WildlifeAfrica/res/values/string.xml</i> . The other way is to type in directly the name like <i>android:title="Refresh"</i>
android:showAsAction	defines how and when the icon does appear

```

1 <menu xmlns:android="http://schemas.android.com/apk/res/android">
2     <item
3         android:id="@+id/action_refresh"
4         android:icon="@drawable/ic_action_refresh"
5         android:title="@string/action_refresh"
6         android:showAsAction="always"/>
7     <item
8         android:id="@+id/action_mapView"
9         android:icon="@drawable/ic_action_map"
10        android:title="@string/action_mapView"
11        android:showAsAction="always"/>
12    <item
13        android:id="@+id/action_filter"
14        android:icon="@drawable/ic_action_search"
15        android:title="@string/action_filter"
16        android:showAsAction="always"/>
17    <item
18        android:id="@+id/action_addSighting"
19        android:icon="@drawable/ic_action_sighting"
20        android:title="@string/action_listView"
21        android:showAsAction="always"/>
22 </menu>

```

Listing 3: Action Bar: Layout SightingListActivity

Implementation

To show the the customized action bar menu, the referenced layout file in the method *onCreateOptionsMenu()* of the *SightingListActivity* must be changed. The listing 4 below shows how.

```
1 getMenuInflater().inflate(R.menu.menu_sighting_list, menu);
```

Listing 4: Action Bar: set the layout

To react to a tap on one of the buttons in the action bar, the activity must override the method *onOptionsItemSelected()*, see listing 5. When this method is called, it receives the clicked item. This item contains an id to identify it. Based on the id, the switch-case command reacts to the click. In this case a notification is shown to the user.

```
1 @Override
2 public boolean onOptionsItemSelected(MenuItem item) {

4     switch (item.getItemId()) {
5     case R.id.action_refresh:
6         Toast.makeText(this, "Refresh", Toast.LENGTH_SHORT).show();
7         break;
8     case R.id.action_mapView:
9         Toast.makeText(this, "Map View", Toast.LENGTH_SHORT).show();
10        break;
11    case R.id.action_filter:
12        Toast.makeText(this, "Filter View", Toast.LENGTH_SHORT).show();
13        break;
14    case R.id.action_addSighting:
15        Toast.makeText(this, "Add Sighting View", Toast.LENGTH_SHORT).
16            show();
17        break;
18    default:
19        break;
20    }
21    return true;
22 }
```

Listing 5: Action Bar: react to click

By default each activity shows the title of the application in the action bar. Disabling the title in the action will provide more space for the icons. Listing 6 shows how to disable the title in the action bar. This code needs to be inside the *onCreate()* method of each activity.

```
1 ActionBar actionBar = getSupportActionBar();
2 actionBar.setDisplayShowTitleEnabled(false);
```

Listing 6: Action Bar: disable title

The following four figures (14, 15, 16 and 17) show the action bars for the four remaining activities. The last three figures (15, 16 and 17) show also an arrow on the left side of the

app icon ,that indicates backward navigation. To enable this arrow, the listing 6 has to be expanded by one line in the respective activities, as shown in the following listing 7.

```
1 actionBar.setDisplayHomeAsUpEnabled(true);
```

Listing 7: Action Bar: enable Home as up

To responds to a tap on the application icon the switch-case command from listing 5 has to be expanded by one case block, shown in listing 8. The id *android.R.id.home* is given by the Android framework.

```
1 case android.R.id.home:
2     Toast.makeText(this, "Back", Toast.LENGTH_SHORT).show();
3     break;
```

Listing 8: Action Bar: react to click on the application icon



Figure 14: Action Bar for the SightingMapActivity



Figure 15: Action Bar for the AddSightingActivity



Figure 16: Action Bar for the SightingFilterActivity













Figure 17: Action Bar for the SightingDetailsActivity

Action Bar Icons

This paragraph describes the icons, that are used for the action bar in the figures 13 to 17. The used icons are provided by Google and they are free to use [Inc13g]. The icon pack for the action bar icons can be downloaded here: <http://developer.android.com/design/downloads/index.html#action-bar-icon-pack>. Table 5 shows the used icons with a short description.

Table 5: Icons for the action bar

Icon	Description
	Access the camera. Open the camera application.
	An arrow, to accept the selection.
	Navigates to the list view.
	Navigates to the map view.
	Synchronizes the local database with a database on a server.
	Navigates to the screen with the filter options.
	Allows to share sighting information via other services.
	Navigates to the screen that allows to add a new sighting.
	Cancels the current process.
	Saves the sighting to the database.

7.6 Activity Layout Files

Each activity has its own XML layout file. This files are stored in *WildlifeAfrica/res/layout*. The following paragraphs summarize the main elements each layout file.

activity_add_sighting.xml contains four *TextView*, two *EditText* to enable the user to write a description and add his name. It defines a *ImageView* to show an image, two *ImageButton*, one to access the camera and the other to access the gallery. And it defines a *Spinner* which enables the user to choose the animal he has seen from a list. The image view uses a icon to indicate an image (see figure 18).



Figure 18: Icon that indicates a picture

activity_sighting_list.xml contains a *TextView*, a *Spinner* and a *ListView*. The *ListView* is for the presentation of the available sightings and the *Spinner* provides a drop down menu to choose a gradation for the list.

activity_sighting_map.xml contains only a *TextView*, that show the text “MAP VIEW”.

activity_sighting_details.xml contains an *ImageView* to show an image and six *TextView* items to show the information about the sighting.

activity_sighting_filter.xml contains a *TextView*, a *ListView* and two *Button* items. The *ListView* shows a list with animal names and check boxes. One *Button* to select all entries in the list, the other deselects all entries.

7.7 AddSightingActivity

This section describes the implementation of the *AddSightingActivity*. This activity enables the user to keep record of new sighting information.

Instance and Class Variables

To reference the UI elements defined in the layout files, this class has for each element, that needs to change in some way, an instance variable, listing 9. These variables are initialized in the *onCreate()* method, shown in the following paragraph.

```
1 private Spinner animalSpinner;
2 private ArrayAdapter<String> spinnerAdapter;
3 private ImageView sightingImage;
4 private ImageButton cameraBtn, galleryBtn;
5 private TextView descriptionView, observerView;
```

Listing 9: AddSightingActivity: instance variables

Methods

AddSightingActivity extends the *Activity* class and therefore the *onCreate()* method has to be overridden, shown in listing 10. The *onCreate()* method is called, when the activity is created for the first time. It has to start with the *super.onCreate()* command in line 3. The following line sets the layout for this activity. Line 6 to 8 shows again how to modify the action bar, like shown in Section 7.5.1. Line 10 to 13 show how to initialize the instance variables with the UI elements defined in the XML layout file (Section 7.6 paragraph “Layout files”). In line 15 a String array is created from a resource file, which are shown in Section 7.4. The lines 18 and 19 show how to insert a list in in the form of a String array into the *Spinner* with the help from an *ArrayAdapter*. The parameter list of the *ArrayAdapter* constructor requires the application context, a layout and the array with the contents to show. As layout file, a standard layout from the Android resource files is chosen. In line 23 the *ArrayAdapter* is set to the *Spinner*. Line 19 to 27 shows how to set a *OnClickListener* to a button. A new *OnClickListener* is created and the *onClick()* method needs to be overridden, to react to the click event. In this case, a small notification is shown on the screen, when the user taps on the camera button. The implementation for the gallery button from line 29 to 36 follows the same pattern.

```

1  @Override
2  protected void onCreate(Bundle savedInstanceState) {
3      super.onCreate(savedInstanceState);
4      setContentView(R.layout.activity_add_sighting);

6      ActionBar actionBar = getActionBar();
7      actionBar.setDisplayShowTitleEnabled(false);
8      actionBar.setDisplayHomeAsUpEnabled(true);

10     animalSpinner = (Spinner) findViewById(R.id.sighting_spinner);
11     sightingImage = (ImageView) findViewById(R.id.sighting_image);
12     descriptionView = (TextView) findViewById(R.id.sighting_description)
        ;
13     observerView = (TextView) findViewById(R.id.sighting_observer);

15     String[] animalEntries = getResources().getStringArray(R.array.
        animalOverview_array);
16     spinnerAdapter = new ArrayAdapter<String>(this, android.R.layout.
        simple_list_item_1, animalEntries);
17     animalSpinner.setAdapter(spinnerAdapter);

19     cameraBtn = (ImageButton) findViewById(R.id.sighting_cameraBtn);

```

```

20     cameraBtn.setOnClickListener(new OnClickListener() {
21         @Override
22         public void onClick(View v) {
23             @Override
24             public void onClick(View v) {
25                 Toast.makeText(getApplicationContext(), "Camera Button",
26                     Toast.LENGTH_SHORT).show();
27             }
28         });

29     galleryBtn = (ImageButton) findViewById(R.id.sighting_galleryBtn);
30     galleryBtn.setOnClickListener(new OnClickListener() {
31         @Override
32         public void onClick(View v) {
33             Toast.makeText(getApplicationContext(), "Gallery Button",
34                 Toast.LENGTH_SHORT).show();
35         }
36     });

```

Listing 10: AddSightingActivity: onCreate() method

The activity also overrides two further methods *onCreateOptionsMenu()* and *onOptionsItemSelected()*, which has been introduced in Section 7.5.1 before in listing 4 and in listing 5.

Navigation

This activity should be able to navigate back to previous screen, regardless of which it was and whether the user saves the sighting information or abort the process (figure 8 in Section 7.1).

The action bar of this contains three clickable elements, the application icon, a trash can, that indicates the abortion of the current process, and a disk icon, that indicates to save the current sighting information. To add the backward navigation functionality, the expansion of the *onOptionsItemSelected()* method is necessary, which was created in Section 7.5.1. Listing 11 shows how to modify the switch-case command to get the backwards navigation. Line 5 shows the call of the private method *saveSighting()*, which contains at this moment only the *super.onBackPressed()* command.

```

1  case android.R.id.home:
2      super.onBackPressed();
3      break;
4  case R.id.action_save:

```

```
5     saveSighting();
6     break;
7 case R.id.action_cancel:
8     super.onBackPressed();
9     break;
```

Listing 11: AddSightingActivity: backward navigation

The implementation for the four activities remaining is very similar, therefore only the relevant differences are documented.

7.8 SightingListActivity

This activity will present a list with all available sightings to the user. This list uses a customized layout. This requires a customized *Adapter* to fill the corresponding *ListView*.

Instance and Class Variables

In addition to the UI elements this activity has also a class variable, that is required for the navigation, to the *SightingFilterActivity*. The usage of this variable is described in the following paragraph “Navigation”. Furthermore this class uses a list with *Sighting* objects, which contains the relevant sighting information. The class *Sighting* is documented in Section 7.12.

```
1 private static final int FILTER_REQUEST = 1;
2 private ListView listView;
3 protected List<Sighting> sightings;
4 private SightingListAdapter listAdapter;
5 private Spinner sortSpinner;
```

Listing 12: SightingListActivity: instance and class variables

Navigation

The action bar contains four elements (see figure 14 in Section 7.5.1. Listing 13 shows how to expand the method *onOptionsItemSelected()*. Outside the switch-case command a new intent is created. Depending on which button is pressed, a new intent with a slightly different parameter list is created. To start an activity via an intent, the intent needs to know the name of the class that should be started, for example in line 7 the second parameter is the name of the *SightingMapActivity*. The activities *SightingMapActivity* and *SightingFilterActivity* are started by calling the method *startActivity* (line 8 and line 16). In this case the current activity is finished and the new one started. The

SightingFilterActivity is started by the command *startActivityForResult*. The difference is, that the *SightingListActivity* will wait for an result from the *SightingFilterActivity*. Section 7.11 describes how to return a result. The second parameter in the method call is the class variable from listing 12. This class variable helps to identify where the call comes from, shown in the next paragraph “onActivityResult”.

```

1  Intent intent = null;
2  switch (item.getItemId()) {
3  case R.id.action_refresh:
4      Toast.makeText(this, "Refresh", Toast.LENGTH_SHORT).show();
5      break;
6  case R.id.action_mapView:
7      intent = new Intent(this, SightingMapActivity.class);
8      this.startActivity(intent);
9      break;
10 case R.id.action_filter:
11     intent = new Intent(this, SightingFilterActivity.class);
12     this.startActivityForResult(intent, FILTER_REQUEST);
13     break;
14 case R.id.action_addSighting:
15     intent = new Intent(this, AddSightingActivity.class);
16     this.startActivity(intent);
17     break;
18 default:
19     break;
20 }
```

Listing 13: *SightingListActivity*: navigation in the action bar

onActivityResult

If an intent is send by the method *startActivityForResult()*, the result is received by the *onActivityResult()* method. This class needs to override the *onActivityResult()* method to handle the result. Listing 14 show the implementation of this method. At first it is necessary to prove whether the *resultCode* is *OK* (line 3) or not (line 9). Then the *requestCode* is checked to identify the start. In line 5 and 6 an *ArrayList* is unpacked from the intent, which is the needed result. How to store data in an intent is shown in Section 7.11.

```

1  @Override
2  protected void onActivityResult(int requestCode, int resultCode, Intent
    data) {
3      if (resultCode == Activity.RESULT_OK) {
```

```

4         if (requestCode == FILTER_REQUEST) {
5             Bundle extra = data.getExtras();
6             ArrayList<String> filterList = (ArrayList<String>) extra.get
                ("list");
7         }
8     }
9     if (requestCode == Activity.RESULT_CANCELED) {
10         if (requestCode == FILTER_REQUEST) {
11         }
12     }
13 }

```

Listing 14: *SightingListActivity*: *onActivityResult()* method

SightingListAdapter

To use a customized list in this activity, it is necessary to create a customized layout file for the list item and to create a customized adapter by generalize the *ArrayAdapter*. All adapter classes are stored in the package *uolnmmu.wildlife.presenter.adapter*. The XML layout file for this list item is the *sighting_list_item.xml* and it is stored with the other layout files. It is created like the layout files for the activities in Section 7.6.

Listing 15 shows the implementation of the *SightingListAdapter* which is used to show, how to fill a *ListView* with customized list entries. The *SightingListAdapter* needs to override the *getView()* method to customize the list item. Line 13 get the clicked line from the list, which is received in the parameter list of the constructor. Line 15 to 23 checks if the needed *View* is present or not. If it is not present, it will be created. Following the UI elements are referenced and then the content is placed. The remaining lines of code reference the UI elements and insert content. As default image the application is shown (line 32).

```

1 public class SightingListAdapter extends ArrayAdapter<Sighting> {
2
3     public SightingListAdapter(Context context, int resource,
4         List<Sighting> sightings) {
5         super(context, resource, sightings);
6     }
7
8     @Override
9     public View getView(int position, View convertView, ViewGroup parent
10        ) {
11         LinearLayout sightingListView;

```

```

12         // get the clicked item
13         Sighting sighting = getItem(position);

14
15         if (convertView == null) {
16             sightingListView = new LinearLayout(getContext());
17             String inflater = Context.LAYOUT_INFLATER_SERVICE;
18             LayoutInflater layoutInflater;
19             layoutInflater = (LayoutInflater) getContext().
                getSystemService(inflater);
20             layoutInflater.inflate(R.layout.sighting_list_item,
                sightingListView, true);
21         } else {
22             sightingListView = (LinearLayout) convertView;
23         }

24
25         // find ui elements
26         ImageView image = (ImageView) sightingListView.findViewById(R.id
            .listItem_photo);
27         TextView animalName = (TextView) sightingListView.findViewById(R
            .id.listItem_name);
28         TextView timestamp = (TextView) sightingListView.findViewById(R.
            id.listItem_date);
29         TextView distance = (TextView) sightingListView.findViewById(R.
            id.listItem_distance);

30
31         // set content to view
32         image.setImageResource(R.drawable.ic_launcher);
33         animalName.setText(sighting.getAnimalName());
34         timestamp.setText(sighting.getTimestamp());
35         if (sighting.getDistance() == -1) {
36             distance.setText("not available");
37         } else {
38             distance.setText(String.valueOf(sighting.getDistance()) + "
                meter");
39         }

40
41         return sightingListView;
42     }
43 }

```

Listing 15: SightingListAdapater: implementation

This *SightingListAdapter* is used in the *onCreate* method to fill the list with sighting information. Listing 16 shows the usage and how to add a *onItemClickListener* and

how to response to a click on a list item. Clicking on a list item triggers to start the *SightingDetailsActivity* (line 20). In line 1 to 6 an *ArrayList* with *Sighting* objects is created to fill the list with first temporary entries. As example the name attribute for the sighting is set. The other attributes can be set as well, but it is not necessary.

```
1  sightings = new ArrayList<Sighting>();
2  for (String animal : getResources().getStringArray(R.array.
    animalOverview_array)) {
3      Sighting s = new Sighting();
4      s.setAnimalName(animal);
5      sightingList.add(s);
6  }

8  listView = (ListView) findViewById(R.id.sightingList_view);

10 // create an ArrayAdapter to fill the ListView
11 listAdapter = new SightingListAdapter(this, R.layout.sighting_list_item,
    sightings);

13 listView.setAdapter(listAdapter);

15 listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
16     Override
17     public void onItemClick(AdapterView<?> parent, View v, int position,
        long id) {
18         Intent intent = new Intent(getApplicationContext(),
19             SightingDetailsActivity.class);
20         startActivity(intent);
21     }
22 });
```

Listing 16: *SightingListActivity*: use *SightingListAdapter*

Sort Sightings

This activity should also provide the possibility to sort the sightings (requirement 2.08, 2.09 and 2.10). To provide a drop down menu, to choose a sort option, a *Spinner* is defined in the XML layout file and implemented in this class, the same way as shown in listing 10 in Section 7.7.

This *Spinner* is additionally provided with an *OnItemSelectedListener*, which calls the respective methods to sort the list. Listing 17. It is determined the position of the clicked element and this position defines, which function to call. The sort-functions are private

function of this class and introduced in the next paragraph “Comparator”. After changing the list, it is necessary to notify the respective adapter (line 7).

```

1  sortSpinner.setOnItemSelectedListener(new OnItemSelectedListener() {
2      @Override
3      public void onItemSelected(AdapterView<?> parent, View view, int
         position, long id) {
4          switch (position) {
5              case 0:
6                  sortSightingsByTime();
7                  listAdapter.notifyDataSetChanged();
8                  break;
9              case 1:
10                 sortSightingsAToZ();
11                 listAdapter.notifyDataSetChanged();
12                 break;
13             case 2:
14                 sortSightingsZToA();
15                 listAdapter.notifyDataSetChanged();
16                 break;
17             case 3:
18                 sortSightingsByDistance();
19                 listAdapter.notifyDataSetChanged();
20                 break;
21         }
22     }
23 }
```

Listing 17: *SightingListActivity*: Spinner *OnItemSelectedListener*

Comparator

A comparator is used to sort objects by specified parameters. In this case, the *ListView* shall be sortable by name (from a to z and from z to a), by time with the last entry at the first position and by distance of the sighting, with the nearest entry at first. Therefore four comparator classes are required, which are stored in the package *uolnmmu.wildlife.presenter.comparator*.

An exemplary implementation for a comparator class is shown in listing 18. The created comparator class needs to implement the interface *Comperator<T>* and override the method *compare()*. In this case the *SightingNameReverseComparator* compares the names of two Sighting objects. The result from the *compareTo()* method in line 15 is reversed by the following if-statements, in order to get a sorting from z to a.

```

1  public class SightingNameReverseComparator implements Comparator<
    Sighting> {

2      @Override
3      public int compare(Sighting lhs, Sighting rhs) {
4          if (lhs.getAnimalName() == null && rhs.getAnimalName() == null)
5              {
6                  return 0;
7              }
8          if (lhs.getAnimalName() == null) {
9              return 1;
10             }
11         if (rhs.getAnimalName() == null) {
12             return -1;
13         }

14         int value = lhs.getAnimalName().compareTo(rhs.getAnimalName());

15         if (value >= 1) {
16             return -1;
17         }
18         if (value <= -1) {
19             return 1;
20         }
21         if (value == 0) {
22             return 0;
23         }

24         return value;
25     }
26 }

```

Listing 18: *SightingListActivity*: *SightingNameReverseComparator*

The *SightingFilterActivity* uses these classes in conjunction with the *java.util.collections* [Ora13a] class, to sort the list with *Sighting* objects. The following listing 19 shows how the *Comparator* from listing 18 is used. The implementation of the other sort functions follow the same pattern.

```

1  private void sortSightingsZToA() {
2      SightingNameReverseComparator comparator = new
        SightingNameReverseComparator();
3      Collections.sort(this.sightings, comparator);
4  }

```

Listing 19: *SightingListActivity*: *sortSightingsZToA()* function

When the activity starts, the list should be sorted by, so that the newest sighting is at the top of the list. To do this, the activity overrides the method *onResume()* which is always called when the activity comes to the foreground. Listing 20 shows how to override the *onResume()* method, how to call of the sort function, and how to notify the adapter, that the list with sighting information changed.

```

1  @Override
2  public void onResume() {
3      super.onResume();

5      sortSightingsByTime();
6      listAdapter.notifyDataSetChanged();
7  }

```

Listing 20: *SightingListActivity*: *onResume()* method

7.9 *SightingMapActivity*

In this iteration, the *SightingMapActivity* is created and filled with a placeholder. It will be modified in Section 9 to display a map and let the user interact with it (requirements 3.02 to 3.08 and 3.10 in Section 4.4).

Layout File At this point, the map will only show a placeholder that can be a *TextView*, that shows the text “MAP VIEW”. The hole XML layout file is shown in listing 21.

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/
    android"
3      android:layout_width="match_parent"
4      android:layout_height="match_parent"
5      android:orientation="vertical" >

7      <TextView
8          android:layout_width="wrap_content"
9          android:layout_height="wrap_content"
10         android:layout_centerHorizontal="true"
11         android:layout_centerVertical="true"
12         android:textSize="30sp"
13         android:text="MAP VIEW" />

15 </RelativeLayout>

```

Listing 21: Layout file: `activity_sighting_map.xml`

Navigation

The implementation of the navigation in the action bar is similar to the implementation in Section 7.8 paragraph “Navigation”. Since the *SightingMapActivity* should also have access to the *SightingFilterActivity* and requires the possibility to get a result from this activity, the implementation is exactly as shown in Section 7.8 paragraph “Navigation” and paragraph ‘onActivityResult’.

7.10 SightingDetailsActivity

This activity displays detailed information about a specific sighting and provides navigation elements. In this iteration, the implementation of this class is limited to showing all created UI elements of the layout file and the action bar, inclusive navigation. The functionality for the share button, in the action bar, is implemented in iteration 4 (Section 9).

Navigation

Like shown in figure 7 this activity should be able to navigate back to the previous screen, navigate to the *AddSightingActivity* and share sighting information. The layout for the action bar is shown in figure 17.

Layout File

The activity fills the UI elements, defined in the XML layout file, with default entries. The default entries are shown in listing 22.

```
1  sightingImage.setImageResource(R.drawable.ic_launcher);
2  nameView.setText("Animal Name");
3  timestampView.setText("yyyy-mm-dd   hh:mm");
4  descriptionView.setText("Some text about the sighting.");
5  observerView.setText("Person who records the sighting.");
```

Listing 22: *SightingDetailsActivity*: set up GUI elements

7.11 SightingFilterActivity

This activity provides a *ListView* which contains *CheckBoxes* and the name of the animals. If a name is clicked the check box is checked or unchecked based on the previous state.

When this activities finishes, by clicking the apply button in the action bar or on the application icon (see figure 16 in section 7.5.1), it returns a list with the selected names to the activity which has started this.

Return a Result

The *FilterSightingActivity* is started with the method *startActivityForResult()* and the activity which has called this functions waits for an result. This result must be defined before the activity finishes. *SightingFilterActivity* contains two private methods. *acceptFilterOptions()* is called, when the apply button in the action bar is clicked and returns a list with animal names as resut. *cancelFilterOptions()* returns the message, that the process was canceled and it is called when the trash can icon in the action bar is clicked. Listing 23 shows the implementation of the *acceptFilterOptions()* method. Inside this method a new intent is created and a new *Bundle* object, which will store the list with names. Line 5 to 10 collects all checked names and stores them in a list. The list is insert to the *Bundle* in line 12, which again is stored in the intent. The result is set to the status *OK* before the activity finishes.

Listing 24 shows the *cancelFilterOptions()* method which set the status of the result to *CANCEL* before the activity finishes.

```

1  private void acceptFilterOptions() {
2      Intent intent = new Intent();
3      Bundle extras = new Bundle();

4
5      ArrayList<String> list = new ArrayList<String>();
6      for (FilterItem item : items) {
7          if (item.isSelected()) {
8              list.add(item.getAnimalName());
9          }
10     }

11
12     extras.putSerializable("list", list);

13
14     intent.putExtras(extras);
15     setResult(Activity.RESULT_OK, intent);
16     this.finish();
17 }
```

Listing 23: SightingFilterActivity: Result OK

```

1  private void cancelFilterOptions() {
2      setResult(Activity.RESULT_CANCELED);
```

```

3         this.finish();
4     }

```

Listing 24: SightingFilterActivity: Result CANCEL

FilterListAdapter

This customized adapter is used for the *ListView* in the *SightingFilterActivity*. Like the *SightingListAdapter*, in Section 7.8, it is a generalization of the *ArrayAdapter* and uses a customized layout file for the list items. The XML layout file for this list item is the *sighting_filter_item.xml* and it is stored with the other layout files. The provided list with animal names is based on the same resource like the list in the *AddSightingActivity* in Section 7.7.

The Implementation and usage of this *FilterListAdapter* follows the same pattern as the *SightingListAdapter* in Section 7.8. Important for the *sighting_filter_item.xml* is, that if check boxes are used in this layout file, they need to specify three certain attributes to avoid problems with *OnClickListeners*. Listing 25 shows this attributes in line 5,6 and 7.

```

1 <CheckBox
2     android:id="@+id/filter_checkBox"
3     android:layout_width="wrap_content"
4     android:layout_height="wrap_content"
5     android:clickable="false"
6     android:focusable="false"
7     android:focusableInTouchMode="false" />

```

Listing 25: FilterListAdapter: layout details

7.12 Sighting

WildlifeAfrica is about documenting, storing and presenting sighting information. That means, that data with sighting information needs to be transferred through this application. In this context the *Sighting* object is a container for this information. It is a simple data transfer object, which only stores data and do not contain any business logic. Figure 19 gives an overview of this class. It only contains private instance variables as well as getter and setter methods for them. The class is stored in the package *uolnmmu.wildlife.model.dataTransferObject*.

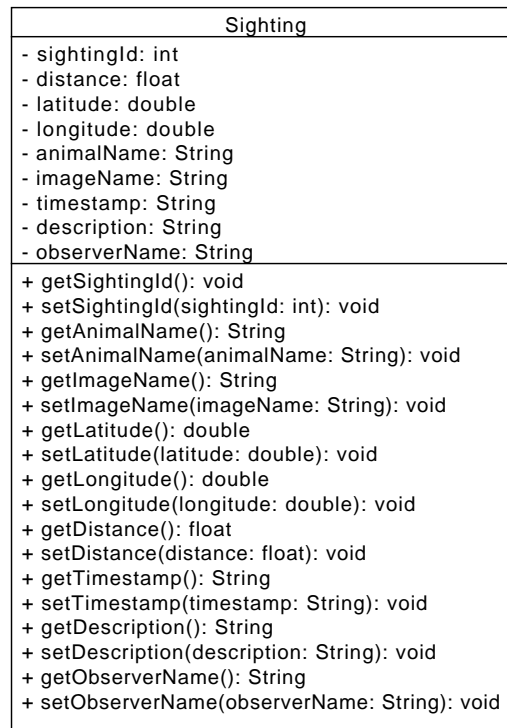


Figure 19: Classdiagram: Sighting class

7.13 Time

This section describes the needed time for this iteration. This can help to decide how to split the tasks and the size of a task for the course. All the work in this iteration took about five hours.

One and a half hour of this time was for the navigation, screens and the mock-ups. This are parts which can take easily a lot of more time because It is a creative work. With pen and paper it is easy to create new ways of navigation but it is difficult to estimate how difficult the implementation is or how long it would take. So this is a first stumbling block in this iteration. Mock-ups with pen and paper are the fastest and easiest way, because tools often tempted to focus to much on details, which cost a lot of time. And time is tight in this intensive course. The layout for the GUI is also a task that can be done quickly with the help of the recommended GUI builder, see Section 3.2. All layout files were created in about 45 minutes. The fastest and often easiest way to define a layout file is by using the GUI builder as well as the XML editor. There are a few commands that can be done with the XML editor much easier as with the GUI builder and conversely

there are tasks, that are easier with the GUI builder. Each course participant has to find his own workflow with the GUI builder and the XML editor. Creating the layout files can be easily divided into smaller task, so that each team of participants can work on on layout file. This approach can give the participants some time to focus on some details if they like and the time allowed it. The implementation of the activities including the action bar, the navigation, the adapter and the comparator took about two and a half hours. Implementing the first adapter can be a little bit tricky and can cost some time, so this could be a small task for one or two course participants. The implementation of the different activities can also be divided between the course participants, to make sure, that all tasks are done at the end of this iteration. Editing the Android Manifest, creating a resource file or the creation of the *Sighting* class are very small and easy tasks, which took about 15 minutes.

8 Iteration 3: Database and Sensors

This chapter describes the implementation of the database and the usage of the sensors, to match the learning outcomes from Section 2.1. The database is required to store and read data. To learn how to use sensors in the Android framework, camera and location functionality are added to the application.

The following section 8.1 describes how to edit the Android manifest, to add the necessary permissions (requirements 1.12, 1.13, 1.14 and 3.01). The next section 8.2 shows how to implement the database layer, how access the database and how to use the created database layer (requirements 1.01 to 1.05 and 1.07 to 1.09). Thereafter follows the usage of the camera 8.5 (requirement 1.12) and the gallery 8.6 (requirement 1.13). The *ImageHelper* class, which is used to simplify the process of storing images, is shown in section 8.4 (requirement 1.06) . This follows the implementation to get the location data 8.7 (requirements 1.15, 1.19, 3.13) and the internet access 8.9. The last section is about the time it took the author, to go through this iteration (Section 8.10).

8.1 Android Manifest

To access the device storage, the camera and the GPS module, some permission entries in the AndroidManifest.xml file are required. All tags have to be inside the *manifest*-tag. The following listing 26 shows the permissions in the Android manifest:

```
1  <!-- camera access -->
2  <uses-feature
3      android:name="android.hardware.camera"
4      android:required="true" />

6  <!-- permission for storage -->
7  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"
8      " />
9  <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"
10     />

12 <!-- permission for location services -->
13 <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"
14     />
15 <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"
16     " />

18 <!-- permission to use the Internet connection -->
19 <uses-permission android:name="android.permission.INTERNET" />
```

```

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

```

Listing 26: AndroidManifest: Permissions to Sensors and Storage

8.2 Database Layer

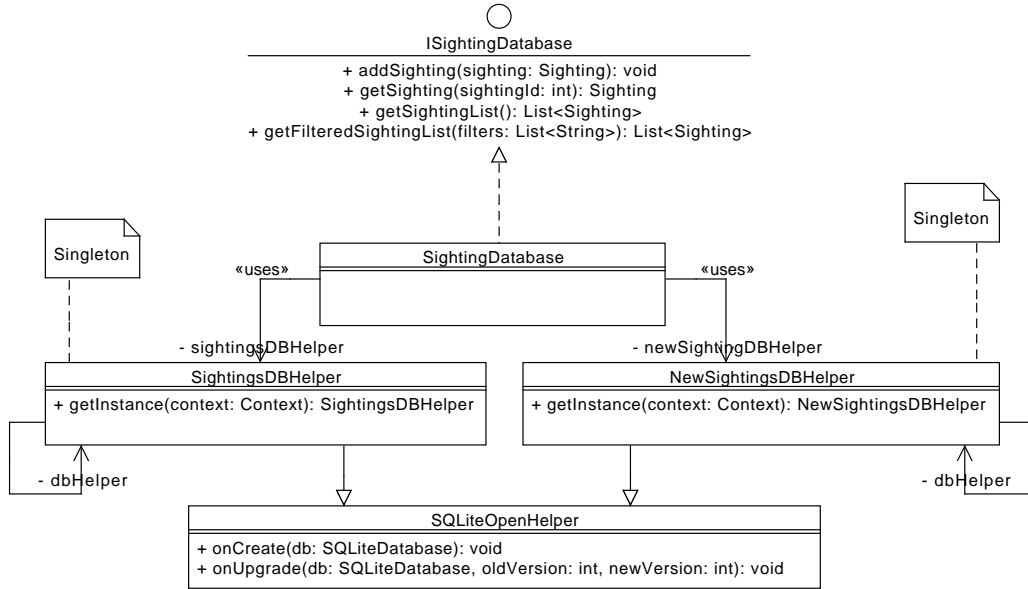


Figure 20: Classdiagram: Database

To store data on the device, the SQLite database from the Android framework is used. In this case the database is used to store information about the sightings. Figure 20 shows the realization of the database layer. The application contains two tables to store data. The design, creation and managing of the database tables is done by the Android framework. All created classes are stored in the package *uolnmmu.wildlife.database*.

One table is named *Sightings* and is managed by *SightingsDBHelper*, the other table is name *NewSightings* and is managed by *NewSightingDBHelper*. Both helper classes generalize the *SQLiteOpenHelper* class which helps to manage the creation of a table. Both classes are singletons to make sure that there is only one object that tries to change, open or close the database. *NewSightings* stores only sightings that were added by the user and *Sightings* stores all available sightings, all that were available on the server and all that were added by the user.

SightingsDatabase operates on the databases and provides functionality to insert or delete data. All necessary SQL statements are created and execute by this class. It realizes at this point the interfaces *ISightingDatabase*.

8.2.1 SightingsDBHelper and NewSightingDBHelper

Both classes create and manage a database table, so that SQL statements can performed on the table. This paragraph describes the implementation of the *SightingsDBHelper*. The implementation for the *NewSightingDBHelper* follows the same pattern.

SightingsDBHelper generalize the class *SQLiteOpenHelper* which is a helper class to interact with a database. Listing 27 shows the constants that are defined as names for the database, the table and the columns inside the table. The column names represent the different information that needs to be stored in the database (requirements 1.01 to 1.04) These constants are used to create an SQL statement. Inside the SQL statement the types for the different values are defined and the *id* is fixed as integer value and the primary key, which helps to identify the sightings. The framework ensures that the primary key is a unique value. At no time the value for the *id* has to be set manually because the framework will increment the value automatically.

```

1 private static SightingsDBHelper dbHelper = null;
2 private static final String DATABASE_NAME = "SightingDatabase";
3 protected static final String TABLE_SIGHTINGS = "Sightings";
4 private static final int DATABASE_VERSION = 1;

6 protected static final String ID = "id";
7 protected static final String ANIMAL_NAME = "animal_name";
8 protected static final String IMAGE_PATH = "image_path";
9 protected static final String GPS_LATITUDE = "gps_latitude";
10 protected static final String GPS_LONGITUDE = "gps_longitude";
11 protected static final String TIMESTAMP = "timestamp";
12 protected static final String DESCRIPTION = "description";
13 protected static final String OBSERVER_NAME = "observer_name";

15 // Create table statement
16 private String CREATE_SIGHTINGS_TABLE = "CREATE TABLE " +
    TABLE_SIGHTINGS
17     + "(" + ID + " INTEGER PRIMARY KEY," + ANIMAL_NAME + " TEXT,"
18     + IMAGE_PATH + " TEXT," + GPS_LATITUDE + " REAL,"
19     + GPS_LONGITUDE + " REAL," + TIMESTAMP
20     + " TEXT," + DESCRIPTION + " TEXT," + OBSERVER_NAME + " TEXT" +
    ")";

```

Listing 27: SightingsDBHelper: column names and create statement

Inside the *SightingsDBHelper* constructor the super method is called to create a new database with a name and a version number, shown in listing 28. The same listing shows the two methods that has to be overridden. The methods *onCreate()* is executed when the database table did not exist, in that case it will create the table with the SQL statement. The method *onUpgrade()* is executed when the version number of the database changes. It deletes the old table and creates a new one.

```

1 private SightingsDBHelper(Context applicationContext) {
2     super(applicationContext, DATABASE_NAME, null, DATABASE_VERSION);
3 }

5 public static SightingsDBHelper getInstance(Context applicationContext)
6     {
7         if (dbHelper == null) {
8             dbHelper = new SightingsDBHelper(applicationContext);
9         }
10        return dbHelper;
11    }

12 @Override
13 public void onCreate(SQLiteDatabase db) {
14     db.execSQL(CREATE_SIGHTINGS_TABLE);
15     Log.d(LOGCAT, "table created");
16 }

18 @Override
19 public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
20     {
21         db.execSQL("DROP TABLE IF EXISTS " + TABLE_SIGHTINGS);
22         onCreate(db);
23     }

```

Listing 28: SightingsDBHelper: methods

8.2.2 SightingsDatabase

To read data and to write data to the database the class *SightingsDatabase* is used. It realizes the functions, that are required through the interface *ISightingDatabase*. This interface is used from all activities, that need access to the database, which is described later in this paragraph.

SightingDatabase must provide functionality to add a new sighting to the database (requirement 1.07), so that the user can store sightings on his device. It must provide functionality to read stored information from the database (requirement 1.09 and requirement 3.09).

To read existing data from a database or to write new data to a database it is necessary to open the database. After the execution of the SQL statements on the database, the database must be closed again. Therefore *SightingDatabase* has some private methods, shown in listing 29. There is a distinction between readable database access and writable database access. If no data in the database needs to be added, updated or deleted the readable access is enough and prevents that data change. If data needs to added, updated or deleted the writable access is required. The following four paragraphs shows how to implement the methods, that are required through the *ISightingDatabase* interface.

```
1 private void openReadableSightingsDB() throws SQLException {
2     sightingsDatabase = sightingsDBHelper.getReadableDatabase();
3 }

5 private void openWritableSightingsDB() throws SQLException {
6     sightingsDatabase = sightingsDBHelper.getWritableDatabase();
7 }

9 private void closeSightingsDB() {
10    sightingsDBHelper.close();
11 }

13 private void openReadableNewSightingDB() throws SQLException {
14     newSightingDatabase = newSightingDBHelper.getReadableDatabase();
15 }

17 private void openWritableNewSightingDB() throws SQLException {
18     newSightingDatabase = newSightingDBHelper.getWritableDatabase();
19 }

21 private void closeNewSightingDB() {
22     newSightingDatabase.close();
23 }
```

Listing 29: SightingDatabase: open and close the database

addSighting(Sighting sighting)

When this method is called it receives a *Sighting* object, that has to be added to the database. This method stores the sighting information in both database tables.

In a first step all information from the *Sighting* are transferred into a *ContentValue* object, which is needed to store the data in the database. This task is done by another private method of *SightingDatabase*, shown in Listing 31. The method returns a *ContentValues* object which contains all information. Following the database for the new sightings is opened, the values are stored and afterwards the database is closed again. The last line shows the call of another private method of *SightingDatabase*. The method receives also the *ContentValues* object and stores the sighting information in the same way to the database, shown in listing 32.

```

1  @Override
2  public void addSighting(Sighting sighting) {
3      ContentValues values = sightingToContentValue(sighting);

5      openWritableNewSightingDB();
6      newSightingDatabase.insert(NewSightingDBHelper.TABLE_SIGHTINGS, null
          , values);
7      closeNewSightingDB();

9      insertSightingToSightingsDB(values);
10 }

```

Listing 30: SightingsDatabase: addSighting

```

1  private ContentValues sightingToContentValue(Sighting sighting) {
2      ContentValues cv = new ContentValues();

4      cv.put(SightingsDBHelper.ANIMAL_NAME, sighting.getAnimalName());
5      cv.put(SightingsDBHelper.IMAGE_PATH, sighting.getImagePath());
6      cv.put(SightingsDBHelper.GPS_LATITUDE, sighting.getGPSLatitude());
7      cv.put(SightingsDBHelper.GPS_LONGITUDE, sighting.getGPSLongitude());
8      cv.put(SightingsDBHelper.TIMESTAMP, sighting.getTimestamp());
9      cv.put(SightingsDBHelper.DESCRPTION, sighting.getDescription());
10     cv.put(SightingsDBHelper.OBSERVER_NAME, sighting.getObserverName());

12     return cv;
13 }

```

Listing 31: SightingDatabase: data into ContentValues

```

1  private void insertSightingToSightingsDB(ContentValues values) {

```

```

2      openWritableSightingsDB();
3      long insertId = sightingsDatabase.insert(SightingsDBHelper.
        TABLE_SIGHTINGS, null, values);
4      closeSightingsDB();
5  }

```

Listing 32: SightingsDatabase: add sighting information

getSighting(int sightingId)

This method returns information about a specific sighting, which is identified by its *sightingId*. Listing 33 shows the implementation of the method. The table is opened with readable access and a new *Sighting* object is created and used to store the information. This object is returned at the end of the method. The *selectQuery* is a String, that represents the SQL statement, which is executed in line 10. The result from the SQL query can be accessed by the *Cursor* interface. The *cursor* is moved to the first entry and the received information are transferred to the *Sighting* object, before the database is closed.

```

1  @Override
2  public Sighting getSighting(int sightingId) {
3      openReadableSightingsDB();
4      Sighting sighting = new Sighting();

6      String selectQuery = "SELECT * FROM "
7          + SightingsDBHelper.TABLE_SIGHTINGS + " WHERE "
8          + SightingsDBHelper.ID + "=" + sightingId;

10     Cursor cursor = sightingsDatabase.rawQuery(selectQuery, null);
11     cursor.moveToFirst();

13     sighting.setSightingId(cursor.getInt(0));
14     sighting.setAnimalName(cursor.getString(1));
15     sighting.setImagePath(cursor.getString(2));
16     sighting.setGPSLatitude(cursor.getDouble(3));
17     sighting.setGPSLongitude(cursor.getDouble(4));
18     sighting.setTimestamp(cursor.getString(5));
19     sighting.setDescription(cursor.getString(6));
20     sighting.setObserverName(cursor.getString(7));

22     cursor.close();
23     closeSightingsDB();
24     return sighting;
25 }

```

Listing 33: SightingsDatabase: getSighting

getSightingList()

This method returns all sighting information, that are stored in the database, so that all available sightings can be shown in list or on a map. The method stores all information, that are received from the database, in an *ArrayList*. This list is returned at the end of the method. The *selectQuery* String asks for all information that are available. The received result is transferred into a list of sighting information with the help of the private method *createSightingList()*, shown in listing 35, before the *cursor* and the database are closed. The method *createSightingList()*, listing 35, loops over all entries in the result, transfers the sighting information into a *Sighting* object and stores this objects in a list, before the list is return.

```

1  @Override
2  public List<Sighting> getSightingList() {
3      openReadableSightingsDB();
4      List<Sighting> sightings = new ArrayList<Sighting>();

6      String selectQuery = "SELECT * FROM "
7          + SightingsDBHelper.TABLE_SIGHTINGS;

9      Cursor cursor = sightingsDatabase.rawQuery(selectQuery, null);

11     sightings = createSightingList(cursor);

13     cursor.close();
14     closeSightingsDB();

16     return sightings;
17 }

```

Listing 34: SightingsDatabase: getSightingList()

```

1  private List<Sighting> createSightingList(Cursor cursor) {

3      List<Sighting> sightings = new ArrayList<Sighting>();

5      if (cursor.moveToFirst()) {
6          do {
7              Sighting sighting = new Sighting();
8              sighting.setSightingId(cursor.getInt(0));

```

```

 9         sighting.setAnimalName(cursor.getString(1));
10         sighting.setImagePath(cursor.getString(2));
11         sighting.setGPSLatitude(cursor.getDouble(3));
12         sighting.setGPSLongitude(cursor.getDouble(4));
13         sighting.setDistance(-1);
14         sighting.setTimestamp(cursor.getString(5));
15         sighting.setDescription(cursor.getString(6));
16         sighting.setObserverName(cursor.getString(7));

18         sightings.add(sighting);
19     } while (cursor.moveToNext());
20 }
21 return sightings;
22 }

```

Listing 35: SightingDatabase: create a list with sightings

getFilteredSightingList(List filters)

When this method is called it receives a list that contains strings. These strings are names from animals, that the user wants to see in *SightingListActivity* or on the *SightingMapActivity* (see 7.6). This matches the requirement 2.03.

The method for this functionality is shown in listing 36. The process is nearly the same as in the listing 35 in the paragraph above. Different is the *selectQuery*, which is in this case slightly more complex. To create this *selectQuery* all Strings from the received list are connected to one String (line 6 to 9). In line 10 the last four character of this *String* are removed. The outcome is used to create the *selectQuery* in line 13. The remaining process is like in the paragraph above.

```

1  @Override
2  public List<Sighting> getFilteredSightingList(List<String> filters) {
3      openReadableSightingsDB();
4      List<Sighting> sightings = new ArrayList<Sighting>();

6      String select = "";
7      for (String entry : filters) {
8          select += SightingsDBHelper.ANIMAL_NAME + "=\"" + entry + "\"" or
          ";
9      }
10     select = select.substring(0, select.length() - 4);

12     // create the hole expression

```

```

13     String selectQuery = "SELECT * FROM " + SightingsDBHelper.
        TABLE_SIGHTINGS + " WHERE " + select;
14     Cursor cursor = sightingsDatabase.rawQuery(selectQuery, null);

16     sightings = createSightingList(cursor);

18     cursor.close();
19     closeSightingsDB();

21     return sightings;
22 }

```

Listing 36: SightingsDatabase: getFilteredSightingList()

8.3 Database Layer Usage

This section describes how to store and read data from the database inside the activities. The activities needs to be expanded, to access the database layer via the *ISightingDatabase* interface.

8.3.1 AddSightingActivity

This activity uses the *SightingsDatabase*, to store new sighting information into the database. Therefore it is necessary to edit and expand this activity.

The information that are collected and stored include the current date and time. To get this information, the activity uses the *SimpleDateFormat* to store the current date and time. It converts it into a String, because the database can only store strings. Listing 37 shows the required instance variables.

```

1 private String timestamp;
2 private SimpleDateFormat sdf;

```

Listing 37: AddSightingActivity: SimpleDateFormat

This two variables are instantiated inside of the *onCreate()* method with the following code lines (listing 8.3.1). *SimpleDateFormat* helps to get the current date and the current time, with a defined layout. The second parameter *Locale.GERMANY* defines the current location.

```

1 sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss", Locale.GERMANY);
2 timestamp = sdf.format(new Date());

```

The method *saveSighting()* is expanded to collect all necessary data, stores them in a *Sighting* object and saves this information into the database. Listing 38 show the method

saveSighting(). Section 8.7 describes how to get the location data, which are listed in line 9 and 10. If no location information are available the process is canceled.

```

1 private void saveSighting() {
2     Sighting sighting = new Sighting();

4     sighting.setAnimalName(animalSpinner.getSelectedItem().toString());
5     if (filename != null) {
6         sighting.setImageName(filename);
7     }
8     if (location != null) {
9         sighting.setLatitude(location.getLatitude());
10        sighting.setLongitude(location.getLongitude());
11        Log.v(LOGCAT, location.getLatitude() + ", " + location.
            getLongitude());
12    } else {
13        Toast.makeText(getApplicationContext(), "No location
            available. Saving canceled", Toast.LENGTH_SHORT).show();
14        return;
15    }
16    sighting.setTimestamp(timestamp);
17    sighting.setDescription(descriptionView.getText().toString());
18    sighting.setObserverName(observerView.getText().toString());

20    ISightingDatabase database = new SightingDatabase(this);
21    database.addSighting(sighting);

23    Toast.makeText(getApplicationContext(), "Saved new Sighting", Toast.
        LENGTH_SHORT).show();

25    super.onBackPressed();
26 }

```

Listing 38: AddSightingActivity: expand saveSighting()

8.3.2 SightingListActivity

The *SightingDatabase* is used to read the available sighting information from the database and show the in a list. In all cases, except when this activity receives a result from the *SightingFilterActivity*, the *SightingList* calls the *getSightingList()* method to get a complete list of sightings. An instance variable is added to hold a reference of *ISightingDatabase*. Listing 39 shows the code snipped from inside the *onCreate()* method, that shows how to get the list with sighting data.

```

1 database = new SightingDatabase(this);
2 sightings = database.getSightingList();

```

Listing 39: SightingListActivity: getSightingList()

The *onResume()* method needs to be expanded, so that the list is updated every time the activity comes to the foreground. For example, when the user returns from adding a new sighting. Because in the Android activity lifecycle the method *onActivityResult()* is executed before the method *onResume()*, it is necessary to check if the activity receives a result from another activity. Therefore the activity holds a instance variable *activityResult*, which is initialized in the *onCreate()* method with false. When the method *onResume()* is called, *activityResult* is set to true. Listing shows the cutout from the *onResume()* method.

```

1 if (this.activityResult == false) {
2     database = new SightingDatabase(this);
3     this.sightings.clear();
4     this.sightings.addAll(database.getSightingList());
5 }
6 sortSightingsByTime();
7 listAdapter.notifyDataSetChanged();
8 this.activityResult = false;

```

Listing 40: SightingListActivity: expand onResume()

The method *onActivityResult* is also expanded in case *SightingListActivity* receives a result from *SightingFilterActivity*. In listing 41 the method *onActivityResult()* first checks the size of the received array. If the array contains elements, this elements are used to get a filtered list of sighting information.

```

1 protected void onActivityResult(int requestCode, int resultCode, Intent
  data) {
2     if (resultCode == Activity.RESULT_OK) {
3         if (requestCode == FILTER_REQUEST) {
4             Bundle extra = data.getExtras();
5             ArrayList<String> filterList = (ArrayList<String>) extra.get(
              "list");
6             if (!filterList.isEmpty()) {
7                 database = new SightingDatabase(this);
8                 this.sightings.clear();
9                 this.sightings.addAll(database.getFilteredSightingList(
                  filterList));
10            }
11            this.activityResult = true;

```



```

12         }
13     }

15     if (requestCode == Activity.RESULT_CANCELED) {
16         if (requestCode == FILTER_REQUEST) {
17             this.activityResult = false;
18         }
19     }
20 }

```

Listing 41: SightingListActivity: expand onActivityResult

8.3.3 SightingMapActivity

The *SightingMapActivity* accesses the database and the *SightingFilterActivity* in the same way the *SightingListActivity* does, shown in Section *db-usage-listActivity*. The only difference is, that this activity does not use the list of sightings at the moment, because the map will be included in iteration four (see also Section 9.2). So at this point, the *SightingMapActivity* can use the database to read information, but it can not present this information to the user.

8.3.4 SightingDetailsActivity

This activity receives a *sightingId* when it is started. This id is used to get the specific sighting information from the database. Listing 42 shows the code snippet from inside the *onCreate()* method. Line 1 and 2 show how to get the id from the intent, that started this activity.

```

1 Bundle extra = getIntent().getExtras();
2 int sightingId = extra.getInt("sightingId");

4 ISightingDatabase database = new SightingDatabase(getApplicationContext()
    ());
5 Sighting sighting = database.getSighting(sightingId);

```

Listing 42: SightingDetailsActivity: get sighting data

So that this activity can receive the id of the specific sighting, the calling activity needs to store this id in the intent, that is used to start this activity. Listing 43 shows exemplary how the *SightingDetailsActivity* is started from the *SightingListActivity* by clicking on a list item. The intent is expanded to store the id.

```

1  Intent intent = new Intent(getApplicationContext(),
    SightingDetailsActivity.class);
2  Sighting sighting = (Sighting) listView.getItemAtPosition(position);
3  int sightingId = sighting.getSightingId();
4  intent.putExtra("sightingId", sightingId);
5  startActivity(intent);

```

Listing 43: Intent to start SightingDetailsActivity

8.4 ImageHelper

The *ImageHelper* class is a small class which has two tasks. The first task is to check if the necessary directory in the device storage exists. The second task is to create files, to store images in this directory. If the directory does not exist, it will be created. This is necessary to store files and images on the device (requirement 1.06 and requirement 1.14). This class is stored in the package *uolnmmu.wildlife.model*.

The *ImageHelper* provides a static method which returns a new file to store an image *getFile(String fileName)* and a static attribute to get the storage location. Listing 44 shows the implementation of the *ImageHelper* class.

```

1  public class ImageHelper {

3      private static final String LOGCAT = ImageHelper.class.getSimpleName();
4      public static final String DIRECTORY = Environment
5          .getExternalStoragePublicDirectory(Environment.
6              DIRECTORY_PICTURES)
7          + File.separator + "Wildlife_Africa" + File.separator;

8      public static File getFile(String fileName) {

10         checkStorageDirectory();

12         File imageFile = new File(DIRECTORY, fileName);
13         try {
14             imageFile.createNewFile();
15         } catch (IOException e) {
16             e.printStackTrace();
17         }
18         return imageFile;
19     }

21     private static void checkStorageDirectory() {

```

```

22         File storageDir = new File(DIRECTORY);

24         if (!storageDir.exists()) {
25             if (!storageDir.mkdir()) {
26                 Log.d(LOGCAT, "Failed to create directory" + DIRECTORY);
27             }
28             Log.d(LOGCAT, "Direktory " + DIRECTORY + " ready to use");
29         }
30     }
31 }

```

Listing 44: ImageHelper: implementation

8.4.1 Usage

Inside the *SightingListAdapter* and the *SightingDetailsActivity* the *ImageHelper* is used to read an image from the internal storage, to show it in a list. Listing 45 shows the code snipped to show a stored image.

```

1  if (sighting.getImageName() != null) {
2      image.setImageDrawable(Drawable.createFromPath(ImageHelper.DIRECTORY
3          + sighting.getImageName()));

```

Listing 45: SightingListAdapter: show image

8.5 Camera

This section describes how to use the default camera application on the Android device with the application *WildlifeAfrica*. The usage of the camera is one requirement from the course as mentioned in Section 2.1 (paragraph 'Databases and Sensors', specified in Section 4.4.1 and listed as requirement 1.12).

The first step, to provide access to the camera, was made with the *permission*-tag in the *Android Manifest* in Section 8.1. Next, the *AddSightingActivity* need to be expanded. If the button for the camera is clicked, an intent will be send to start the default camera app on the device. To receive the taken photo, the camera app will also return an intent. The *AddSightingActivity* has to override the method *onActivityResult*, to react to the camera response and to receive the intent. Listing 46 shows the intent that will be created and send when the camera button is clicked. This piece of source code is inside the *onClickListener* of the camera button. The intent contains some extras, to specify the image, that will be returned. In this case the image will be cropped and scaled down to 450px to 450px.

Line 12 specified the location, to save the arranged image in a new location. The location is specified by an *Uri*, which is provided by the *ImageHelper* class, see Section 8.4. The *Intent* is send in line 14 by the method *startActivityForResult()*. The second parameter is an integer, that helps to identify the call in *onActivityResult()*. It is a class variable, shown in listing 47.

```

1  filename = "IMG_" + timestamp + ".jpg";
2  File file = ImageHelper.getNewFile(filename);
3  imageUri = Uri.fromFile(file);

5  Intent takePictureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
6  takePictureIntent.putExtra("crop", "true");
7  takePictureIntent.putExtra("outputX", 450);
8  takePictureIntent.putExtra("outputY", 450);
9  takePictureIntent.putExtra("aspectX", 1);
10 takePictureIntent.putExtra("aspectY", 1);
11 takePictureIntent.putExtra("scale", true);
12 takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT, imageUri);

14 startActivityForResult(takePictureIntent, CAMERA_REQUEST);

```

Listing 46: AddSightingActivity: start the camera app

```

1  private static final int CAMERA_REQUEST = 1;

```

Listing 47: AddSightingActivity: CAMERA_REQUEST

The method *onActivityResult* receives three information, when the activity comes back to the foreground. The *requestCode* that helps to identify where the call comes from, the *resultCode* to identify if the call was successful or not and an intent with further information based on the call. The *requestCode* and the *resultCode* must be checked before continuing with the process.

```

1  if (resultCode == Activity.RESULT_OK) {
2      if (requestCode == CAMERA_REQUEST) {

4          bitmap = BitmapFactory.decodeFile(imageUri.getPath());
5          sightingImage.setImageBitmap(bitmap);

7          // Save the image to gallery. Create an Intent
8          Intent mediaScanIntent = new Intent(Intent.
          ACTION_MEDIA_SCANNER_SCAN_FILE);
9          mediaScanIntent.setData(imageUri);

11         // send a Broadcast to notyfiy the Gallery

```

```

12         this.sendBroadcast(mediaScanIntent);
13     }
14 }

```

Listing 48: Handle the result from camera request

Listing 48 shows how to check the result from the camera request. First, the *resultCode* is checked to make sure that the request was successful and an image is available. Second, identify the request based on the *requestCode*. If both checks are successful, the new image is grabbed from the storage directory and shown in the GUI. Google recommends to make all images available through the default gallery. The lines 7 to 12 in listing 48 show how to notify the gallery to search in a specified directory for new images, so that they can be shown in the default gallery.

8.6 Gallery

To receive an image from the Android gallery, is very similar to the way, to receive an image from the camera in section 8.5. Also in this case, the gallery is started by sending an intent. This intent also contains further information about how to handle the image. As well the result is available in the *onActivityResult* method in the *AddSightingActivity*.

```

1  Intent takePictureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
2  // put Extras into the intent and launch the camera
3  takePictureIntent.putExtra("crop", "true");
4  takePictureIntent.putExtra("outputX", 450);
5  takePictureIntent.putExtra("outputY", 450);
6  takePictureIntent.putExtra("aspectX", 1);
7  takePictureIntent.putExtra("aspectY", 1);
8  takePictureIntent.putExtra("scale", true);
9  takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT, imageUri);

11 startActivityForResult(takePictureIntent, CAMERA_REQUEST);

```

Listing 49: Intent to start the gallery app

Listing 49 shows the intent, that is used to start the gallery and to specify the image to return. The additional parameter for this Intent are the same as for the camera in section 8.5. The second parameter *GALLERY_REQUEST* in the method from line 11 is again an Integer to identify the call, see listing 50.

```

1  private static final int GALLERY_REQUEST = 2;

```

Listing 50: AddSightingActivity: gallery request

How to handle the answer in *onActivityResult* is shown in listing 51. If the *resultCode* and the *requestCode* are fine, the image can be shown in the GUI.

```

1  if (resultCode == Activity.RESULT_OK) {
2      ...
3      if (requestCode == GALLERY_REQUEST) {
4          bitmap = BitmapFactory.decodeFile(imageUri.getPath());
5          sightingImage.setImageBitmap(bitmap);
6      }
7  }

```

Listing 51: Handle the result from gallery request

8.7 LocationService

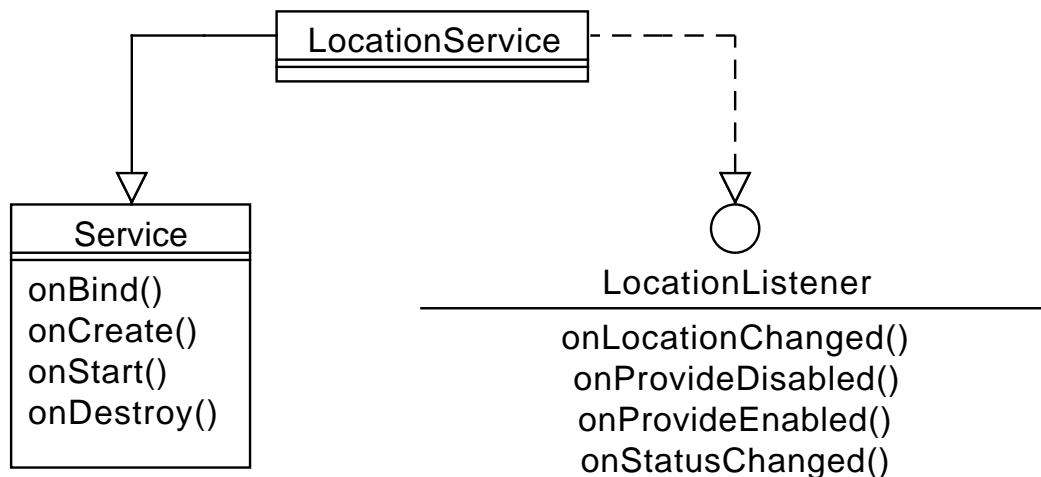


Figure 21: Classdiagram: LocationService

The application needs data, to represents the current location of the user. This data is used to describe the location of an animal sighting and in the following iteration, to show the position of the user on a map (see Section 9.2.4).

Figure 21 shows the composition of the *LocationService* class, which is a background service, that provides location information. To work in the background it must generalize the *Service* class. To get information about the current location, it must imple-

ment the *LocationListener* interface. The created class is stored in the package *uolnmmu.wildlife.model.gpsService*.

Listing 52 shows how to override the *onStart()* method. When the service is created successful this method is called. First, it is necessary to get the *LocationManager* from the *SystemService*. Line 3 and 4 show how to create *criteria*, to define how precise the location data should be and which information are necessary. The *LocationManager* will choose the best available service provided by this criteria. This can be the GPS sensor or the localization via the network connection. Line 8 requests a single location update. This way saves battery, and for this application, it is not necessary to request the location information in planned intervals.

```

1  locationManager = (LocationManager) getSystemService(serviceName);

3  Criteria criteria = new Criteria();
4  criteria.setAccuracy(Criteria.ACCURACY_COARSE);

6  String provider = locationManager.getBestProvider(criteria, true);

8  locationManager.requestSingleUpdate(provider, this, null);

```

Listing 52: LocationService: ask for location data

The location data are received via the *onLocationChanged()* method, which must be overridden, because it is required by the *LocationListener* interface. The method received a *Location* object, which contains the latitude and longitude information. Listing 53 shows, that these received information are send directly via an intent. In this case a *Broadcast* is send, to notify all activities within this application.

```

1  @Override
2  public void onLocationChanged(Location location) {
3      intent.putExtra("location", location);
4      intent.setAction(BROADCAST_ACTION);
5      sendBroadcast(intent);
6  }

```

Listing 53: LocationService: receive location data

8.8 LocationService Usage

The activities, which should be able to receive this information, must register a *BroadcastReceiver*. For this application the *SightingListActivity* and *AddSightingActivity* needs this location data. So these two activities have to be extended by an Broadcast receiver, which

must be registered and deregistered during the lifecycle of the activity. Furthermore, both activities should start the *LocationService*, when they come to the foreground.

8.8.1 Android Manifest

To use a service that works in the background, it has to be registered to the Android-Manifest.xml file. The *service*-tag has to be inside the *application*-tag. It is important to enter the exact path to the service class, like shown in listing 54 below.

```
1 <service
2     android:name="uolnmmu.wildlife.model.gpsService.LocationService"
3     android:enabled="true" >
```

Listing 54: Manifest: register LocationService

When an activity uses a *BroadcastReceiver* it is also necessary to register an *intent-filter*, so that the activities know, which broadcast intent is relevant. Listing 55 shows the snippet that stands inside the *activity*-tag of the *SightingListActivity* and the *AddSightingActivity*.

```
1 <intent-filter>
2     <action android:name="LocationService" />
3 </intent-filter>
```

Listing 55: Manifest: intent-filter

8.8.2 Start Location Service

To start the location service, both mentioned activities need to expand the *onResume()* method, so that, every time the activity comes to the foreground, the location information is updated. Listing 56 shows how to start the location service, using an intent inside of the *onResume()* method.

```
1 Intent service = new Intent(this, LocationService.class);
2 startService(service);
```

Listing 56: SightingListActivity: start location Service

8.8.3 Broadcast Receiver

Listing 57 shows how to add a new *BroadcastReceiver* to an activity. The *BroadcastReceiver* listens for intents, that are directed to this activity. When it receives an intent, it is unpacked, to work with the provided information. The location data is shown to the

user by a notification. Both activities, *AddSightingActivity* and *SightingListActivity*, call a setter method *setLocation(newLocation)*, shown in listing 58, to set the value from the intent to the instance variable from type *Location*. The activity *SightingListActivity* also calls the the method *calcDistanceForSightings()* to update the distance information.

```

1 private BroadcastReceiver locationReciver = new BroadcastReceiver() {

3     @Override
4     public void onReceive(Context context, Intent intent) {

6         Location newLocation = (Location) intent.getExtras()
7             .get("location");
8         Toast.makeText(
9             getApplicationContext(),
10            newLocation.getLatitude() + ", "
11            + newLocation.getLongitude(), Toast.
12                LENGTH_SHORT)
13            .show();

14            setLocation(newLocation);
15            calcDistanceForSightings();
16        }
17    };

```

Listing 57: *SightingListActivity*: add a *BroadcastReceiver*

```

1 protected void setLocation(Location newLocation) {
2     this.location = newLocation;
3 }

```

Listing 58: *SightingListActivity*: *setLocation()*

The method *calcDistanceForSightings()*, called in listing 57, calculates the difference between two locations. In this case, the one location is the location of the user and the other location is the location of the sighting. The *Location* class provides a *distanceTo()* method, to calculate the distance. The implementation for the *calcDistanceForSightings()* method is shown in listing 59. For each sighting in the list of sightings, the difference is calculated. The latitude and longitude value are stored in a new *Location* object to use the *distanceTo()* method, which calculates the distance. After updating all sightings with new distance information the list in the *SightingListAdapter* is updated.

```

1 private void calcDistanceForSightings() {
2     ArrayList<Sighting> list = new ArrayList<Sighting>();
3     if (location != null) {

```

```

4         for (Sighting sighting : this.sightings) {

6             double sLat = sighting.getLatitude();
7             double sLong = sighting.getLongitude();

9             Location sightingLocation = new Location((String) null);
10            sightingLocation.setLatitude(sLat);
11            sightingLocation.setLongitude(sLong);

13            float distance = this.location.distanceTo(sightingLocation);

15            sighting.setDistance(distance);
16            list.add(sighting);
17        }
18        this.listAdapter.clear();
19        this.listAdapter.addAll(sightings);
20        this.listAdapter.notifyDataSetChanged();
21    }
22 }

```

Listing 59: SightingListActivity: calcDistanceForSightings()

8.8.4 Manage Broadcast Receiver

A *BroadcastReceiver* needs to be registered to an activity, when the activity comes to the foreground and it needs to be deregistered, when an activity goes to the background, to make sure, that the *BroadcastReceiver* can be used by the activity. A *BroadcastReceiver* is registered inside of the *onResume()* method, because this method is called every time the activity comes to the foreground. To register a *BroadcastReceiver* an *IntentFilter* is needed, to define the service.

```

1  IntentFilter locationFilter = new IntentFilter();
2  locationFilter.addAction(LocationService.BROADCAST_ACTION);
3  registerReceiver(locationReceiver, locationFilter);

```

Listing 60: SightingListActivity: register the BroadcastReceiver

When an activity goes to the background, the method *onPause()* is called, so that the *BroadcastReceiver* is deregistered in this method. Listing 61 shows how to override the *onPause()* method and how to deregister the *BroadcastReceiver* from the activity (line 4).

```

1  @Override
2  public void onPause() {
3      super.onPause();

```

```

4     unregisterReceiver(locationReceiver);
5 }

```

Listing 61: SightingListActivity: deregister the BroadcastReceiver

8.9 Internet

WildlifeAfrica needs access to the internet in order to get information about the actual position, to access the map and to upload and download data from an external database. The application has only access to all required resources if the permissions are declared in the AndroidManifest.xml file. This happens before in iteration 1 in section 7.3.

Furthermore, it is important to check if an internet connection is available, before starting an upload or download. The upload and download of data can be triggered from the *SightingListActivity* and the *SightingMapActivity*. Like shown in listing 62 is the *ConnectionManager* used to check, if an connection is available. The user is notified whether there is an internet connection or not.

```

1  ConnectivityManager connMgr = (ConnectivityManager) getSystemService(
        Context.CONNECTIVITY_SERVICE);
2  NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();

4  if (networkInfo != null && networkInfo.isConnected()) {
5      Toast.makeText(getApplicationContext(), "Network connection is
        available.", Toast.LENGTH_SHORT).show();
6  } else {
7      Toast.makeText(getApplicationContext(), "No network connection
        available.", Toast.LENGTH_SHORT).show();
8  }

```

Listing 62: Check if internet connection is available

8.10 Time

The learning outcomes in Section 2.1 mentioned for this iteration that the course participants are able to use the database and different sensors. The implementation of the database layer took some time but with some basic knowledge in SQL it can be done. The whole implementation took nearly six hours.

The implementation of the database layer, Section 8.2, and the link to the activities took about three and a half hours. This is a task that is not easy to divide in smaller task. When the interface for the database layer is defined first, on task can be creating the database layer and another task can be to link the activities with the interface to the

database layer. Editing the activities cost a lot of time. Implementing the *ImageHelper* class, the camera and the gallery can be done in about 45 minutes up to one hour. The *LocationService*, Section 8.7, is a task that took about one and a half hour. The stumbling block is the concept of the *Service* and the *BroadcastReceiver* because it is easy to forget to register something in the Android Manifest or to send a wrong intent when the service is done. Checking for a available internet connection, Section 8.9, is a task that cost only some minutes.

9 Iteration 4: External Services

This chapter refers to the learning outcomes from Section 2.1 paragraph “External Services”. To match the topics for this iteration, the participants have to integrate and use the Google Maps Services, share information about a sighting with other services like Facebook or Flickr and they have to synchronize the local database on the Android device with an external database on a server.

The first section describes the external database, which is required for the synchronization (Section 9.1). The implementation for the required functionality of the application is shown in the following section. Section 9.2 shows the implementation for Google Maps and how to display customized information on the map, this refers to the requirements 3.02, 3.03, 3.05, 3.06, 3.07, 3.10 from Section 4.4. Section 9.3 shows how to integrate functionality to share information with other applications (refers to requirements 2.07 and 4.02). Afterwards the Section 9.4 shows the implementation of the *SyncService* which provides functionality to synchronize the local database with a database on a server (requirements 1.10 and 1.11). The last section is about the time it took the author to go through this iteration (Section 9.5).

9.1 External Database

This section describes the external database. The database stores sighting information and provides this information to users, who synchronizes their local database with the external database. For an intensive course it would be the best, if the database is set up by the teaching staff, so that the course participants can focus on developing the Android application.

The external database is a MySQL database. The database *SightingsDatabase* contains a table *Sighting*, shown in table 6. The *ID* is the primary key to identify the sighting, furthermore it is an integer, that increments itself automatically. The column *image* needs to be from type *mediumblob*, to store images. *gps_latitude* and *gps_longitude* store values from type double, to store the location information as exact as possible. All other values are stored as text.

Table 6: MySQL Database: Table

#	Name	Type	Key	Extra
1	ID	int	Pri	auto_increment
2	animal_name	text		
3	image_path	text		
4	gps_latitude	double		
5	gps_longitude	double		
6	timestamp	text		
7	description	text		
8	observer_name	text		
9	image	mediumblob		

To access the database, the database needs an interface, which provides two functionalities. It must be able to receive JSON data, to add this new data to the database and it must be able to provide JSON data with all available sighting information in the database. The author uses to small PHP scripts to provide this interface, but other solutions are possible, as long as JSON is used to transfer the data.

The PHP script, to add new sighting information, is shown in listing 63. The script contains the information that are required to set up the connection to the database. It opens the database table and inserts the received data. The second script, to get all sighting data from the database, is shown in listing 64. This script also contains the information to connect to the database. It connects to the database, opens the table to read the data from the database and stores them in a JSON container.

```

1  <?
2  // read JSon input
3  $result = json_decode(file_get_contents('php://input'));

5  $hostname = "127.0.0.1";
6  $username = "root";
7  $password = "";
8  $db_name = "SightingsDatabase";

10 // try to connect to the database
11 $con=mysql_connect("$hostname", "$username", "$password") or die ("
    cannot connect");
12 mysql_select_db("$db_name") or die ("cannot select database");

```

```

14 foreach($result as $key => $value) {
15     mysql_query("INSERT INTO SightingsDatabase.Sighting (animal_name,
        image_path, gps_latitude,
16         gps_longitude, timestamp, description, observer_name, image)
17         VALUES ('$value->animal_name', '$value->image_path', '$value->
            gps_latitude', '$value->gps_longitude',
18             '$value->timestamp', '$value->description', '$value->
                observer_name', '$value->image')");
19 }

21 mysql_close($con);
22 ?>

```

Listing 63: PHP script: addSighting.php

```

1  <?
2  $hostname = "127.0.0.1";
3  $username = "root";
4  $password = "";
5  $db_name = "SightingsDatabase";

7  // try to connect to the database
8  $con=mysql_connect("$hostname", "$username", "$password") or die ("
        cannot connect");
9  mysql_select_db("$db_name") or die ("cannot select database");

11 $sql = "SELECT * FROM Sighting";
12 $result = mysql_query($sql);
13 $json = array();

15 if(mysql_num_rows($result)){
16     while($row=mysql_fetch_assoc($result)){
17         //$json['Sightings'][]=$row;
18         $json[]=$row;
19     }
20 }

22 print json_encode($json);
23 mysql_close($con);
24 ?>

```

Listing 64: PHP script: getAllSightings.php

9.2 GoogleMaps

To provide a view that show a map the user can interact with, several steps are necessary. Several permissions needs to be added to the Android manifest, to enable the application to use Google Maps (Section 9.2.1). The *SightingMapActivity* layout file requires only a small modification to show the map (Section 9.2.2). The implementation of the main functionality is show in the Sections 9.2.3 to 9.2.5.

9.2.1 Manifest

To use Google Maps in this application several permissions have to be declared in the Android Manifest. First the application needs the permission to receive a map from a server. This *permission*-tag is shown in Listing 65. Furthermore two *uses-permission*-tags are required, so that the received map can be used (line 5) and the application can use the Google Maps API (line 6).

```
1 <permission
2     android:name="uolnmmu.wildlife.permission.MAPS_RECEIVE"
3     android:protectionLevel="signature" />

5 <uses-permission android:name="uolnmmu.wildlife.permission.MAPS_RECEIVE"
6     />
7 <uses-permission android:name="com.google.android.providers.gsf.
8     permission.READ_GSERVICES" />
```

Listing 65: Manifest: permission for maps

To draw a map Android uses OpenGL ES, so that this feature needs to be activated in the AndroidManifest.xml file. Listing 66 shows the required *uses-feature*-tag.

```
1 <uses-feature
2     android:glEsVersion="0x00020000"
3     android:required="true" />
```

Listing 66: Manifest: OpenGL ES

The last entry is for the required API-key. The *meta-data*-tag, shown in listing 67 needs to be inside the *application*-tag. The API-Key in line 3 is required to access the Google Maps server. The shown key is just an example and won't work in an application. The API-key must be created for each Project on each machine, that is used for developing, because it is based on a digital certificate, stored on the computer. Google provides a detailed guidance on this page: https://developers.google.com/maps/documentation/android/start#getting_the_google_maps_android_api_v2.


```

1 <meta-data
2     android:name="com.google.android.maps.v2.API_KEY"
3     android:value="AIzaSyDtVf6czpoEOKLvGEc_COVymEBNPHh7CrU" />

```

Listing 67: Manifest: Google Maps

9.2.2 Layout File

The layout file for the *SightingMapActivity* must be expanded by a *fragment* that will display the map. Listing 68 shows the *fragment*-tag that needs to be insert to the layout file. Line 4 is important, because it defines what to show in the *fragment*.

```

1 ...
2 <fragment
3     android:id="@+id/map_view"
4     android:name="com.google.android.gms.maps.MapFragment"
5     android:layout_width="match_parent"
6     android:layout_height="match_parent" />
7 ...

```

Listing 68: SightingMap Layout: Fragment for the Map

9.2.3 Initialize the Map

To show the map on the screen, the *SightingMapActivity* is expanded by a *initializeMap()* method, which checks if a map is already created or if it needs to be created. This method is called in the *onCreate()* method and the *onResume()* method of the *SightingMapActivity*.

Because the layout file defines a *fragment*, it is necessary to access this fragment by a *FragementManager* (line 3). This *FragementManger* locates the defined *fragment* and gets the map from it, which is stored as a instance variable called *googleMap*. This is the main object, which provides the map functionality and will be used for all other following modification.

```

1 private void initilizeMap() {
2     if (googleMap == null) {
3         googleMap = ((MapFragment) getFragmentManager().findFragmentById(
4             R.id.map_view)).getMap();
5
6         if (googleMap == null) {
7             Toast.makeText(getApplicationContext(), "Unable to create maps",
8                 Toast.LENGTH_SHORT).show();
9         }
10    }

```

```
8 }
```

Listing 69: SightingMapActivity: initialize the map

9.2.4 Camera and Position

To view the own position on the map, it requires only a few lines of code. In the *initializeMap()* method, from the paragraph above, the second if-statement (line 5) is expanded by an else-statement. Inside the else-statement the function *setMyLocationEnabled* from the *googleMap* object is called (listing 70). The camera for the start point of the map is also inside the else-statement. Line 3 to 6 shows how to define the entry point for the camera.

```
1 ...
2 } else {
3     // Kragga Kamma Game Park
4     LatLng target = new LatLng(-33.985860, 25.458548);

6     googleMap.animateCamera(CameraUpdateFactory.newLatLngZoom(target,
7         15));

8     googleMap.setMyLocationEnabled(true);
9 }
```

Listing 70: SightingMapActivity: show own position

9.2.5 Marker on the Map

Like the *SightingListActivity* this activity needs access to the database. It receives a list from the *SightingFilterActivity* to filter the sighting data. It also inserts the id into an intent, to start the *SightingDetailsActivity*. The implementation for this functionality is similar to the implementation for the *SightingListActivity*.

To set the marker on the map, the *SightingMapActivity* contains a private method *setMarkerOnMap()*, to set a marker for every entry of the list with sightings. Listing 71 shows the *setMarkerOnMap()* method. It uses a *HashMap* to map the marker with the sightings. This is necessary to get the right *sightingId* when the user tabs on the marker. This method is called in the *onResume()* method and the *onActivityResult()* method to place the marker.

```
1 private void setMarkerOnMap() {
2     googleMap.clear();
3     sightingMarkerMap = new HashMap<Marker, Sighting>();
```

```

5      for (Sighting sighting : sightingsList) {
6          marker = googleMap.addMarker(new MarkerOptions().position(new
              LatLng(sighting.getLatitude(), sighting.getLongitude()))
7              .title(sighting.getAnimalName())
8              .snippet(sighting.getTimestamp()));

10         sightingMarkerMap.put(marker, sighting);
11     }
12 }

```

Listing 71: SightingMapActivity: setMarkerOnMap()

9.2.6 OnInfoWindowClickListener

When the user tabs a maker on the map, an info window shows up which, provides further information about the sighting. So that the application can show the *SightingDetailsActivity*, when the info window is clicked, a *OnInfoWindowClickListener* needs to be registered to the *GoogleMap* object. The *else*-statement from Section 9.2.4 is expanded by the code snippets in the listing 72. The sighting to the corresponding marker is read from the *HashMap*. The *sightingId* from the sighting is stored in an intent, which is used to start the *SightingDetailsActivity* exactly as in the *SightingListActivity*.

```

1  googleMap.setOnInfoWindowClickListener(new OnInfoWindowClickListener() {

3      @Override
4      public void onInfoWindowClick(Marker arg0) {
5          Sighting sighting = sightingMarkerMap.get(marker);
6          Intent intent = new Intent(getApplicationContext(),
              SightingDetailsActivity.class);
7          int sightingId = sighting.getSightingId();
8          intent.putExtra("sightingId", sightingId);

10         startActivity(intent);
11     }
12 });

```

Listing 72: SightingMapActivity: OnInfoWindowClickListener

9.3 Sharing

In Section 2.1 paragraph 'External Services' is mentioned, that the course participants shall learn how to share information from their application with other services. The Android

framework provides an easy way to share information and data with other applications by using *Intents*, which are shortly introduced in Section 6.5.

To share the information that are displayed in the *SightingDetailsActivity* with other applications, the activity needs to be expanded by the function shown in listing 73. This method is called, when the user taps on the share button in the action bar (see Figure 17). In this case, the message consists of a text, which is just the animal's name and the image of the animal. In line 7 a new intent is created and following filled with the text message and the image. Line 12 starts an activity, which shows a screen with a list of applications, that can receive this intent. By clicking on one of the list items, the respective application starts, receives the intent and shows the content of the intent in the context of the respective application. That makes it easy to share information about a sighting with a lot of services by using their application.

```

1  private void startSharing() {
2      String message = nameView.getText().toString();

4      File imageFile = ImageHelper.getNewFile(imageName);
5      Uri uri = Uri.fromFile(imageFile);

7      Intent sharingIntent = new Intent(Intent.ACTION_SEND);
8      sharingIntent.setType("*/*");
9      sharingIntent.putExtra(android.content.Intent.EXTRA_TEXT, message);
10     sharingIntent.putExtra(android.content.Intent.EXTRA_STREAM, uri);

12     startActivity(Intent.createChooser(sharingIntent, "Share this
        Sighting"));
13 }

```

Listing 73: SightingDetailsActivity: share functionality

9.4 SyncService

The *SyncService* consists of two classes, that provides the functionality to synchronize the SQLite database on the Android device, with a MySQL database on a server. The class *UploadService* (Section 9.4.2) provides functionality to upload data to the MySQL database and the class *DownloadService* (Section 9.4.3) provides functionality to download data. To exchange data between the Android application and the server the JSON format is used. Android includes all required libraries by default. The classes are stored in the package *uolnmmu.wildlife.syncService*.

Both services uses the *IUpdateSightingDatabase* interface, which allows to write and

delete data from the database. This interface has to be created and the class *SightingDatabase* expanded. The implementation is shown in Section 9.4.1.

9.4.1 IUpdateSightingDatabase

The interface *IUpdateSightingDatabase* defines methods which writes and deletes data from the database. It is implemented by the class *SightingDatabase* from Section 8.2.2, so that this class needs to be expanded. *IUpdateSightingDatabase* defines three methods which implementations are shown in the next paragraphs.

getSightingsFromNewSightingDB

This method returns a list with all sightings, that are stored in the table *NewSightings*. The implementation shown in listing 74 is very similar to the implementation of the method *getSightingList()*. The only difference is, that in this implementation the *NewSightingDBHelper* is used and the *NewSightings* database is opened and closed.

```

1  @Override
2  public List<Sighting> getSightingsFromNewSightingDB() {
3      openReadableNewSightingDB();

5      List<Sighting> sightings = new ArrayList<Sighting>();
6      String selectQuery = "SELECT * FROM " + NewSightingDBHelper.
          TABLE_SIGHTINGS;
7      Cursor cursor = newSightingDatabase.rawQuery(selectQuery, null);
8      sightings = createSightingList(cursor);

10     cursor.close();
11     closeNewSightingDB();

13     return sightings;
14 }
```

Listing 74: SightingDatabase: getSightingsFromNewSightingDB()

updateSightingDatabase(List sightingList)

This method receives a list with sightings (listing 75). All current entries from the database are deleted by calling the method *deleteAllSightings()*. The implementation is shown in listing 76. Each entry from the received list is converted into a *ContentValues* object and stored into the database (line 5 to 8).

```

1  @Override
2  public void updateSightingDatabase(List<Sighting> sightingList) {
3      ContentValues contentValues;
4      deleteAllSightings();

6      for (Sighting sighting : sightingList) {
7          contentValues = sightingToContentValue(sighting);
8          insertSightingToSightingsDB(contentValues);
9      }
10 }

```

Listing 75: SightingDatabase: updateSightingDatabase()

```

1  private void deleteAllSightings() {
2      openWritableSightingsDB();
3      String selectQuery = "DELETE FROM " + SightingsDBHelper.
4          TABLE_SIGHTINGS;
5      sightingsDatabase.execSQL(selectQuery);
6      closeSightingsDB();
7  }

```

Listing 76: SightingDatabase: deleteAllSightings

clearNewSightingsDatabase()

The implementation of this method, shown in listing 77, is very similar to the implementation of the method *deleteAllSightings* shown in listing 76 in the paragraph above.

```

1  @Override
2  public void clearNewSightingsDatabase() {
3      openWritableNewSightingDB();
4      String selectQuery = "DELETE FROM " + NewSightingDBHelper.
5          TABLE_SIGHTINGS;
6      newSightingDatabase.execSQL(selectQuery);
7      closeNewSightingDB();
8  }

```

Listing 77: SightingDatabase: clearNewSightingsDatabase()

9.4.2 UploadService

The *UploadService* is a background task, which uploads data to a database. So that the *UploadService* can handle asynchronous requests and can work in the background, it is necessary that this class extends the class *IntentService*. This is a service, that handles

requests on demand and is started by an intent. *UploadService* defines some instance and some class variables, shown in listing 78. *URL* is the url to the server and the PHP script, which handles the request. The *HttpClient* handles the connection management, the *HttpPost* is used to send data to the server. Furthermore this class holds a *ArrayList* with *Sighting* objects and can access the database via the *IUpdateSightingDatabase*.

```

1 private final String URL = "http://192.168.2.105/wildlifeAfrica/
  addSightings.php";

3 private HttpClient client = new DefaultHttpClient();
4 private HttpPost httpPost = new HttpPost(URL);

6 private ArrayList<Sighting> sightingList;
7 private IUpdateSightingDatabase database;
```

Listing 78: UploadService: variables

Because *UploadService* extends *IntentService*, it has to override the methods *onCreate()*, *onDestroy()* and *onHandleIntent()*. Listing 79 shows from line 1 to line 19 the implementation of this three methods. *onCreate()* is called when the service is created, it fetches all sightings, that are made by the user, from the database. *onDestroy()* is called when the service is no longer used. When this services ends it calls the method *notifyDownloadService()*. This method is shown in line 84 to 89. It creates an intent to start the *DownloadService*. The important part happens in the *onHandleIntent()* method. This method tries to upload the data to the database and clears the local database afterwards. To upload the data, it calls the method *uploadDataToServer()*, which is shown from line 21 to 37. First the list with *Sighting* objects is converted into a *JSONArray* with the help from the method *sightingListToJSON()* (line 39 to 61). This method creates for each *Sighting* object a *JSONObject*, which will store the data from the *Sighting* object. To store the image, the method *encodeImageFromPath()* converts the image into a *String* (line 62 to 82). When the image is converted into a *String* and all data are stored in the *JSONObject*, the *uploadToServer()* method tries to upload the data to the server (line 24 to 31).

```

1 @Override
2 public void onCreate() {
3     super.onCreate();
4     Toast.makeText(getApplicationContext(), "Upload Started", Toast.
      LENGTH_SHORT).show();
5     database = new SightingDatabase(this);
6     sightingList = (ArrayList<Sighting>) database.
      getSightingsFromNewSightingDB();
```

```
7  }

9  @Override
10 public void onDestroy() {
11     super.onDestroy();
12     notifyDownloadService();
13 }

15 @Override
16 protected void onHandleIntent(Intent intent) {
17     uploadDataToServer();
18     database.clearNewSightingsDatabase();
19 }

21 private void uploadDataToServer() {
22     JSONArray jsonArray = sightingListToJSON(sightingList);

24     try {
25         StringEntity entity = new StringEntity(jsonArray.toString());

27         entity.setContentType(new BasicHeader(HTTP.CONTENT_TYPE, "
            application/json"));
28         this.httpPost.setEntity(entity);

30         HttpResponse response = client.execute(httpPost);

32     } catch (ClientProtocolException e) {
33         e.printStackTrace();
34     } catch (IOException e) {
35         e.printStackTrace();
36     }
37 }

39 private JSONArray sightingListToJSON(ArrayList<Sighting> sightingList) {
40     JSONArray array = new JSONArray();

42     for (Sighting s : sightingList) {
43         JSONObject object = new JSONObject();

45         String encodedImage = encodeImageFromPath(s.getImageName());
46         try {
47             object.put("animal_name", s.getAnimalName());
48             object.put("image_path", s.getImageName());
49             object.put("gps_latitude", s.getLatitude());
```



```

50         object.put("gps_longitude", s.getLongitude());
51         object.put("timestamp", s.getTimestamp());
52         object.put("description", s.getDescription());
53         object.put("observer_name", s.getObserverName());
54         object.put("image", encodedImage);
55     } catch (JSONException e) {
56         e.printStackTrace();
57     }
58     array.put(object);
59 }
60 return array;
61 }

63 private String encodeImageFromPath(String imagePath) {
64     File imageFile = null;

65
66     try {
67         imageFile = new File(ImageHelper.DIRECTORY, imagePath);
68     } catch (Exception e) {
69         e.printStackTrace();
70     }

71
72     if (imageFile != null && imageFile.exists()) {
73         Bitmap bitmap = BitmapFactory.decodeFile(imageFile.
74             getAbsolutePath());
75         ByteArrayOutputStream baos = new ByteArrayOutputStream();
76         bitmap.compress(Bitmap.CompressFormat.JPEG, 100, baos);
77         byte[] byteArray = baos.toByteArray();

78         return Base64.encodeToString(byteArray, Base64.DEFAULT);
79     }

80
81     return null;
82 }

84 private void notifyDownloadService() {

85
86     Toast.makeText(getApplicationContext(), "Upload Complete", Toast.
87         LENGTH_SHORT).show();
88     Intent dlService = new Intent(this, DownloadService.class);
89     startService(dlService);
90 }

```

Listing 79: UploadService: methods

9.4.3 DownloadService

The implementation of the *DownloadService* is similar to the implementation of the *UploadService*. Listing 80 shows the class and instance variables. The String *Broadcast_Action* is used to mark the intent, which will be send when this service completes the task. The *URL* is the url to the target script on a server. The *StringBuilder* in line 4 is used to read the received JSON String from the server. The *HttpClient* manages the connection to the server and the *HttpGet* is the method to get the data from the server. This class also access the database via the *IUpdateSightingDatabase*, to store sighting information.

The *onCreate()* method creates the intent and creates a new database.

```

1 public static final String BROADCAST_ACTION = "DownloadService";
2 private final String URL = "http://192.168.2.100/wildlifeAfrica/
  getAllSightings.php";

4 private StringBuilder builder = new StringBuilder();
5 private HttpClient client = new DefaultHttpClient();
6 private HttpGet httpGet = new HttpGet(URL);
7 private Intent intent;
8 private IUpdateSightingDatabase database;
```

Listing 80: DownloadService: variables

This class also extends the class *IntentService* and has to override the methods *onCreate()* and *onHandleIntent()*. Listing 81 shows the implementation of all relevant methods. The *onHandleIntent()* method calls successively the methods *downloadData()*, which will download new sighting data from the server, *saveToDatabase()* which saves the new data to the SQLite database and *broadcastResult()* which sends an intent as a broadcast, so that all involved activities get notified. *downloadData()*, from line 16 to 35, executes the *HttpGet*, reads out and checks the status code of the answer. When there is a correct answer the content is stored in the *StringBuilder*.

saveToDatabase(), from line 37 to 63, converts the content from the *StringBuilder* into a String and uses this to create a *JSONArray*. Line 43 runs through the array to read out each single *JSONObject*. This object is used to read out the detailed sighting data, so that this data can be stored in a *Sighting* object. The *Sighting* objects are stored in a list, which is used to store new sighting data into the database (line 61).

In line 54 the method *storeImage()* (line 64 to 82) is called to store the received image. In a first step the String is decoded into a Byte array, which is used to create a new Bitmap. This is used to write it into a new file (line 71 to 74). Line 75 calls the method *notifyGallery()* which sends an intent so that the Android Gallery searches in the given

location for a new image.

The last method called by *onHandleIntent()* is the method *broadcastResult()* (line 91 to 97). This method creates an intent and sends it as broadcast to notify all involved activities.

```

1  @Override
2  public void onCreate() {
3      super.onCreate();
4      Toast.makeText(getApplicationContext(), "Download Started", Toast.
        LENGTH_SHORT).show();
5      database = new SightingDatabase(this);
6      intent = new Intent(BROADCAST_ACTION);
7  }

9  @Override
10 protected void onHandleIntent(Intent intent) {
11     downloadData();
12     saveToDatabase();
13     broadcastResult();
14 }

16 private void downloadData() {
17     try {
18         HttpResponse response = client.execute(httpGet);
19         StatusLine statusLine = response.getStatusLine();
20         int statusCode = statusLine.getStatusCode();
21         if (statusCode == 200) {
22             HttpEntity entity = response.getEntity();
23             InputStream content = entity.getContent();
24             BufferedReader reader = new BufferedReader(new
                InputStreamReader(content));
25             String line;
26             while ((line = reader.readLine()) != null) {
27                 builder.append(line);
28             }
29         }
30     } catch (ClientProtocolException e) {
31         e.printStackTrace();
32     } catch (IOException e) {
33         e.printStackTrace();
34     }
35 }

37 private void saveToDatabase() {

```

```
38     List<Sighting> list = new ArrayList<Sighting>();

40     JSONArray jsonArray;
41     try {
42         jsonArray = new JSONArray(builder.toString());
43         for (int i = 0; i < jsonArray.length(); i++) {
44             Sighting sighting = new Sighting();
45             JSONObject jsonObj = jsonArray.getJSONObject(i);

47             sighting.setAnimalName(jsonObj.getString("animal_name"));
48             sighting.setImageName(jsonObj.getString("image_path"));
49             sighting.setLatitude(jsonObj.getDouble("gps_latitude"));
50             sighting.setLongitude(jsonObj.getDouble("gps_longitude"));
51             sighting.setTimestamp(jsonObj.getString("timestamp"));
52             sighting.setDescription(jsonObj.getString("description"));
53             sighting.setObserverName(jsonObj.getString("observer_name"));
54             ;
55             storeImage(jsonObj.getString("image"), jsonObj.getString("image_path"));

56             list.add(sighting);
57         }
58     } catch (JSONException e) {
59         e.printStackTrace();
60     }
61     database.updateSightingDatabase(list);
62 }

64 private void storeImage(String encodedImage, String imageName) {

66     byte[] byteArray = Base64.decode(encodedImage, Base64.DEFAULT);
67     Bitmap bitmap = BitmapFactory.decodeByteArray(byteArray, 0,
68         byteArray.length);

69     File imageFile = ImageHelper.getNewFile(imageName);
70     try {
71         FileOutputStream fileOut = new FileOutputStream(imageFile);
72         bitmap.compress(Bitmap.CompressFormat.JPEG, 100, fileOut);
73         fileOut.write(byteArray);
74         fileOut.close();
75         notifyGallery(Uri.fromFile(imageFile));

77     } catch (FileNotFoundException e) {
78         e.printStackTrace();
79     }
80 }
```

```

79     } catch (IOException e) {
80         e.printStackTrace();
81     }
82 }

84 private void notifyGallery(Uri uri) {
85     Intent mediaScanIntent = new Intent(Intent.
86         ACTION_MEDIA_SCANNER_SCAN_FILE);
87     mediaScanIntent.setData(uri);

88     this.sendBroadcast(mediaScanIntent);
89 }

91 private void broadcastResult() {

93     Toast.makeText(getApplicationContext(), "Download Complete", Toast.
94         LENGTH_SHORT).show();
95     intent.putExtra("string", builder.toString());
96     intent.setAction(BROADCAST_ACTION);
97     sendBroadcast(intent);
98 }

```

Listing 81: DownloadService: methods

9.4.4 BroadcastReceiver

The activities *SightingListActivity* and *SightingMapActivity* have both to register a *BroadcastReceiver*, to receive the message, that the upload and download are finished. Listing 82 shows how to create the *BroadcastReceiver* inside the *SightingMapActivity*. The Implementation for the *SightingListActivity* is very similar, only the handling with the list of sightings changes, shown in listing 83.

```

1 private BroadcastReceiver downloadReceiver = new BroadcastReceiver() {
2     @Override
3     public void onReceive(Context context, Intent intent) {

5         SightingMapActivity.this.sightingsList = (ArrayList<Sighting>)
6             SightingMapActivity.this.database.getSightingList();
7         SightingMapActivity.this.setMarkerOnMap();
8     }
9 };

```

Listing 82: SightingMapActivity: create the BroadcastReceiver

```

1 private BroadcastReceiver downloadReciver = new BroadcastReceiver() {
2     @Override
3     public void onReceive(Context context, Intent intent) {

5         SightingListActivity.this.sightings = (ArrayList<Sighting>)
            SightingListActivity.this.database.getSightingList();

7         SightingListActivity.this.listAdapter.clear();
8         SightingListActivity.this.listAdapter.addAll(sightings);
9         SightingListActivity.this.listAdapter.notifyDataSetChanged();
10    }
11 };

```

Listing 83: SightingListActivity: create BroadcastReceiver

Each *BroadcastReceiver* has to be registered in the *onResume()* method and deregistered in the *onPause()* method. Listing 84 shows the code snipped inside the *onResume()* method, that registered the *BroadcastReceiver*. Listing 85 shows how to deregister the *BroadcastReceiver* inside the *onPause()* method. This is in both activities the same.

```

1 IntentFilter updateFilter = new IntentFilter();
2 updateFilter.addAction(DownloadService.BROADCAST_ACTION);
3 registerReceiver(downloadReciver, updateFilter);

```

Listing 84: SightingMapActivity: register the BroadcastReceiver

```

1 unregisterReceiver(downloadReciver);

```

Listing 85: SightingMapActivity: deregister the BroadcastReceiver

9.4.5 Android Manifest

The services needs to be register in the AndroidManifest.xml file, like the *LocationService* from Section 8.7. Listing 86 shows the required tags which stands inside the *application-*tag.

```

1 <service
2     android:name="uolnmmu.wildlife.model.syncService.UploadService"
3     android:enabled="true" >
4 </service>
5 <service
6     android:name="uolnmmu.wildlife.model.syncService.DownloadService"
7     android:enabled="true" >
8 </service>

```

Listing 86: Manifest: UploadService and DownloadService

Inside the tags of *SightingListActivity* and *SightingMapActivity* an *intent-filter*-tag is required, because both classes uses a *BroadcastReceiver*. Listing 87 shows the required tag.

```
1 <intent-filter>
2     <action android:name="LocationService" />
3 </intent-filter>
```

Listing 87: Manifest: intent-filter

9.5 Time

In this iteration Google Maps was used to show and work with a map, the application was expanded by sharing functionality and the application is now able to synchronize the internal database with a database on a server. The implementation of this functionality took about four hours and 45 minutes.

Implementing and using the map to show sighting information is a task that can be done in about one and a half hour. The tricky part is to get the API-Key, which is different for each used computer. If the participants work with their own computers, than they have to get the API-Key by their own, which can coast some additional time. If the participants work with given computers, it would be the best, if they can get the required SHA-1 fingerprint for their computer from the teaching staff, because it save time and sometimes also nerves. When the author tried to implement the map, Google had some server problems, so that he was not able to register a valid API-Key for the application. If something like this happens again, there is no way to get the map working, because the API-Key is necessary to access the Google Maps server. The implementation for the sharing functionality can be done in a few minutes.

The *SyncService* took a little bit longer with three hours. The difficult part of this implementation is inside the *onHandleIntent* methods, where it is necessary to write files to the storage or convert images. Because the concept of the implementation for both service classes is similar, this task can be split up, so that the participants work on this tasks on the same time.

10 Conclusion

This bachelor's thesis runs as part of a project of the University of Oldenburg (Germany) and the Nelson Mandela Metropolitan University (South Africa). Both universities planned and created a mobile computing course. This is a one week intensive course with the goal to provide knowledge about designing, developing and evaluating mobile applications for mobile platforms like Android, iOS or Windows Phone. This knowledge will be consolidated by developing an Android application. In an intensive course is not much time, so an Android application is required, which can be developed as a part of the course. The goal of this bachelor's thesis was to design and develop an Android application for a mobile computing course, which covers the topics of the course and the development process fits into the course schedule.

At first this bachelor's thesis describes the framework for a mobile computing course to arrange the different topics to different iterations (Section 2). In the context of the intensive course, each day corresponds to one day. To unify and support the development process, different development tools are used. These tools and some alternatives are presented in Section 3. Following Section 3.4 shows how to set up these tools. Section 4 describes the requirements for the Android application, which are used to define the functional range of the application. The section provides an application vision (Section 4.1), use cases (Section 4.3) and user stories with listed requirements (Section 4.4). The following section describes the architecture of the application (Section 5). Each iteration has one section, in which the development process of the application is described and documented. Iteration "Android Framework" in Section 6, iteration "GUI" (Section 7), iteration "Database and Sensors" (Section 8) and iteration "External Services" (Section 9). Each of these sections ends with a paragraph that mentions the time it took the author to pass through the iteration and which stumbling blocks may occur. This paragraph also validates that the application can be developed within these iterations.

The designed and developed Android application is matched to the course schedule and can be developed in four iterations. The functionality of the application is separated into different modules, to add the functionality step by step. This allows, to split the development work from each iteration in smaller tasks. That makes the application scalable. The task can be allocated on all course participants, if there is not much time. Splitting up the application into smaller modules enables this application to work as a sample. In case, that course participants can not solve a task on their own, they can get the needed module from the teaching staff to continue with the next task or the next iteration. This application uses a game park sighting context to cover the different course topics. This

context can be changed easily. If no game park or similar is available, the context can be changed to search, document and share information about local cars, buildings, plants or something else. If the context fits into the topics of this course, it can be used as context for the application.

11 Outlook

The present application gives an overview of the Android framework and shows how to work with it. It do not cover all aspects and concepts of developing for Android, which gives starting points for a further development.

Developing for mobile devices means to develop for various of screen sizes and various of screen resolution. In a further development, this Android application can be expanded to support a variety of screen sizes, to push the importance of the user experience more in the foreground. Other extensions can include more sensors like environmental sensors to read out the current temperature or to measure the current brightness, when the user documents a sighting [[Inc13l](#)]. It is important to keep in mind, that not all devices include all sensors. Another interesting extension can be the usage of further communication techniques like NFC (Near Field Communication) [[Inc13k](#)] or bluetooth [[Inc13f](#)]. This techniques allow to transfer data on a short range between devices. NFC is relatively new technology, that is, up to now, not used in many devices. Depending on the application scenario, it can be interesting to include other media types like audio and video.

12 CD Content

The CD contains the bachelor’s thesis as a PDF file, the folder *workspace-android* and the folder *PHP Scripts*. The folder *workspace-android* contains for each iteration an Android project. Furthermore it contains an Android project for the “Hello World” application and an Android library-project for the Google Play Services library. Figure 22 shows this folder structure on the cd.

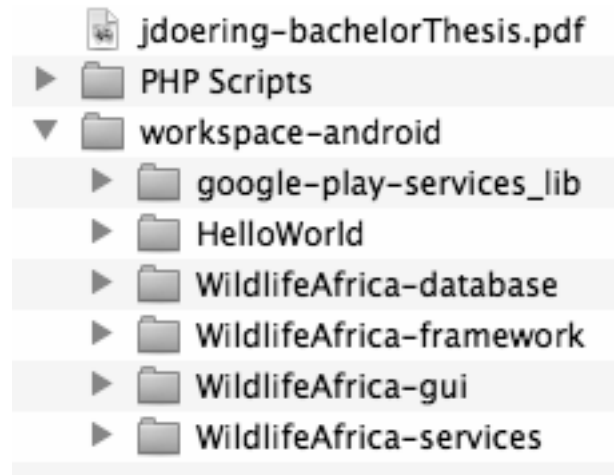


Figure 22: Folder structure on the cd

13 List of References

References

- [Ax13] ANDROID-X86: *Android-x86 - Porting Android to x86*. <https://sites.google.com/a/android-x86.org/web/>. Version: Oct 2013. – Website
- [BMZ13] BMZ: *Federal Ministry for Economic Cooperation and Development*. <http://www.bmz.de/en/index.html>. Version: Jun 2013. – Website
- [Bra97] BRADNER, S.: Key Words for use in RFCs to Indicate Requirement Levels. In: *Network Working Group, Category: Best Current Practice* (1997). <http://www.ietf.org/rfc/rfc2119.txt>
- [DAA13] DAAD: *German Academic Exchange Service*. <https://www.daad.de/en/>. Version: Jun 2013. – Website
- [das13] *Developing and Strengthening Industry-driven Knowledge-transfer between developing Countries*. : *Developing and Strengthening Industry-driven Knowledge-transfer between developing Countries*, Jun 2013. <http://www.dasik.org/>. – Website
- [Dro13] DROIDDRAW: *DroidDraw Beta*. <http://www.droiddraw.org/>. Version: Jul 2013. – Website
- [Fow13] FOWLER, Martin: *GUI Architectures*. <http://www.martinfowler.com/eaDev/uiArchs.html>. Version: Aug 2013. – Website
- [Gen13] GENYMOTION: *Genymotion*. <http://www.genymotion.com/>. Version: Sep 2013. – Website
- [HB12] HOOBER, S. ; BERKAMN, E.: *Designing Mobile Interfaces*. Sebastopol : O'Reilly Media, Inc., 2012
- [Inc13a] INC., Google: *Action Bar*. <http://developer.android.com/guide/topics/ui/actionbar.html>. Version: Sep 2013. – Website
- [Inc13b] INC., Google: *Android Developer Tools*. <http://developer.android.com/tools/help/adt.html>. Version: Jul 2013. – Website
- [Inc13c] INC., Google: *Android Emulator*. <http://developer.android.com/tools/help/emulator.html>. Version: Oct 2013. – Website

- [Inc13d] INC., Google: *Application Fundamentals*. <http://developer.android.com/guide/components/fundamentals.html>. Version: Sep 2013. – Website
- [Inc13e] INC., Google: *AVD Manager*. <http://developer.android.com/tools/help/avd-manager.html>. Version: Jul 2013. – Website
- [Inc13f] INC., Google: *Bluetooth*. <http://developer.android.com/guide/topics/connectivity/bluetooth.html>. Version: Oct 2013. – Website
- [Inc13g] INC., Google: *Downloads*. <http://developer.android.com/design/downloads/index.html>. Version: Sep 2013. – Website
- [Inc13h] INC., Google: *Get the Android SDK*. <http://developer.android.com/sdk/index.html>. Version: Jul 2013. – Website
- [Inc13i] INC., Google: *Getting Started with Andorid Studio*. <http://developer.android.com/sdk/installing/studio.html>. Version: Jul 2013. – Website
- [Inc13j] INC., Google: *Google Play Services*. <http://developer.android.com/google/play-services/index.html>. Version: Oct 2013. – Website
- [Inc13k] INC., Google: *Near Field Communication*. <http://developer.android.com/guide/topics/connectivity/nfc/index.html>. Version: Oct 2013. – Website
- [Inc13l] INC., Google: *Sensors Overview*. http://developer.android.com/guide/topics/sensors/sensors_overview.html. Version: Oct 2013. – Website
- [Inc13m] INC., Google: *Using Hardware Devices*. <http://developer.android.com/tools/device.html>. Version: Oct 2013. – Website
- [Kle11] KLEUKER, S.: *Grundkurs Software Engineering mit UML*. 2. Edition. Wiesbaden : Vieweg + Teubner Verlag, 2011
- [Mei12] MEIER, R.: *Professinal Android 4 Application Development*. Indianapolis : John Wiley & Sons, Inc., 2012
- [Ora13a] ORACLE: *Class Collections*. <http://docs.oracle.com/javase/1.4.2/docs/api/java/util/Collections.html>. Version: Sep 2013. – Website
- [Ora13b] ORACLE: *VirtualBox*. <https://www.virtualbox.org/>. Version: Jul 2013. – Website

- [OUO13a] OSSIETZKY UNIVERSITÄT OLDENBURG, Carl von: *Algorithmen und Datenstrukturen*. <http://www-ui.informatik.uni-oldenburg.de/46628.html>. Version: Aug 2013. – Website
- [OUO13b] OSSIETZKY UNIVERSITÄT OLDENBURG, Carl von: *Algorithmen und Programmierung*. <http://www-ui.informatik.uni-oldenburg.de/1442.html>. Version: Aug 2013. – Website
- [OUO13c] OSSIETZKY UNIVERSITÄT OLDENBURG, Carl von: *Informationssysteme 1*. <http://www-is.informatik.uni-oldenburg.de/52/>. Version: Aug 2013. – Website
- [OUO13d] OSSIETZKY UNIVERSITÄT OLDENBURG, Carl von: *Java Programmierkurs*. <http://www-is.informatik.uni-oldenburg.de/~dibo/teaching/programmierkurs.html>. Version: Aug 2013. – Website
- [OUO13e] OSSIETZKY UNIVERSITÄT OLDENBURG, Carl von: *Softwaretechnik 1*. <http://www.se.uni-oldenburg.de/preprocess.php?seite=45624.html&include0=generated-content/2011-se.html&clearAll>. Version: Aug 2013. – Website
- [Par13] PARK, Kragga Kamma G.: *Welcome to Kraggakamma*. <http://www.kraggakamma.com/content.asp>. Version: Aug 2013. – Website
- [Sta12] STAUEMEYER, J.: *Android Programmierung*. Köln : O'Reilly Verlag, 2012
- [Tec13] TECHNOLOGY, Massachusetts I.: *MIT App Inventor*. <http://appinventor.mit.edu/explore/>. Version: Jul 2013. – Website
- [Tra13a] TRAVEL, Satpack: *Africa: Live*. <https://play.google.com/store/apps/details?id=com.kruger.live.working>. Version: Aug 2013. – Website - GooglePlay Store
- [Tra13b] TRAVEL, Satpack: *Africa: Live*. <https://itunes.apple.com/us/app/kruger-live/id552879842?ls=1&mt=8>. Version: Aug 2013. – Website - iTunes
- [Tra13c] TRAVEL, Satpack: *Africa: Live*. <http://www.wildafricalive.com/>. Version: Aug 2013. – Website

-
- [WCW⁺13] WESSON, J. L. ; COWLEY, N. L. O. ; WINTER, A. ; PETERS, D. ; PRESENTERS, Unspecified I. ; PRESENTERS, Unspecified M.: *Mobile Computing (MC) Draft Course Schedule*. Jun 2013. – internal document
- [You90] YOURDON, E.: *Modern Structured Analysis*. New Jersey : Prentice-Hall, 1990

Abschließende Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Oldenburg, den October 30, 2013

Julian Döring