System Design Document for StudyBuddy

Bashar Oumari, Benjamin Sannholm, Mathias Drage,
Pontus Nellgård, Sophia Pham

2020-10-21
version 2

1 Introduction

The project is a mobile application developed for Android smartphones using Android Studio. The main purpose of the app is to allow students to locate and interact with other students registered to the same courses and universities who want to study together or are looking for new study partners. The idea is loosely based on a Snapchat extension app called "Beer Buddy" in which people can see other friends and what they are currently drinking and request to join them. This document provides a high-level overview of the implementation of the application.

1.1 Definitions, acronyms, and abbreviations

Term	Definition
Activity	A user interface "scene" in which the current session is logged and the user can interact with different Android UI components.
Android Studio [1]	The framework used for developing the GUI, logic, and running the Android phone emulator.
Broadcast	The team's technical term for the users to display that they are actively seeking a study partner. It holds information about the course, description, and the location of the study session (Broadcast). On the map, they are seen as pins.
Firebase [2]	A Google-developed "backend as a service" that works well with the Google Maps location integration and account setup.
Fragment	Similar to an Activity (scene) in which the user can interact with and navigate between different Android components (UI).
Jetpack Navigation [3]	A framework for Android development to make it easier to navigate between different UI screens.
GPS	A self-developed class to handle geolocation and requesting

	the user's device (and the user) to use geolocation activities (longitude and latitude).
Pin (see also "dot on the map")	A red Google Maps icon that displays where a specific broadcast is located.
Google Maps [4]	A framework provided by Google to use their Maps API in Android applications
MVC	Model View Controller, a design pattern
OOP	Object-Oriented Programming, a design pattern
GUI	Graphical User Interface
JUnit [5]	A framework used for testing
Espresso test recorder [6]	An android framework used for testing android UI interactions.
Gradle [7]	A framework for managing dependencies
Git [8]	A version control manager
GitHub [9]	A platform for managing projects using Git
AVD	Android Virtual Device, an android emulator mocking an android smartphone.
JaCoCo [10]	Java Code Coverage, a tool for Java which generates a code coverage report.
STAN [11]	A tool for creating dependency graphs.

2 System architecture

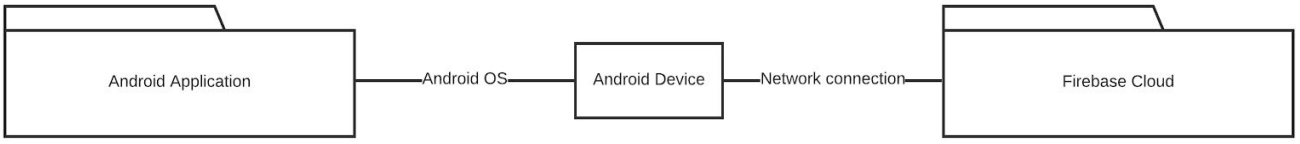

Figure 1. *System components.*

Firebase is responsible for receiving and storing information about "broadcasts" created by different instances of the application from different users, and distributing this to the other instances of the program. Firebase also has the responsibility for storing user information, more specifically authentication credentials for each user that has registered on the application. This is so the user is able to authenticate themselves so the application can keep track of which broadcast is created by which user. The protocols used to communicate with

Firebase are encapsulated in the Firebase client libraries and is not something we had to concern ourselves about.

The Android device is the device the application is running on. The device provides information like GPS location and current time to the application. The device also has a network connection which is necessary for the application to communicate with Firebase.

The Android application has responsibility for all interaction between the application and the user of the application, like views and user inputs. It has the functionality for the application to work, and functionality to get necessary information from the Android device it is running on. When the application creates a broadcast, it starts an "Android service" on the device, which keeps the broadcast alive by updating the broadcast's last active date, and sends an updated version of the broadcast to Firebase when changes are made. The network connection between the device and Firebase also lets the application receive information of broadcasts created by other instances of the application, and show them on the view in the application. No information about broadcasts is persisted locally. Once the application is closed all information about broadcasts is persisted in Firebase and is retrieved again when needed.

3 System design

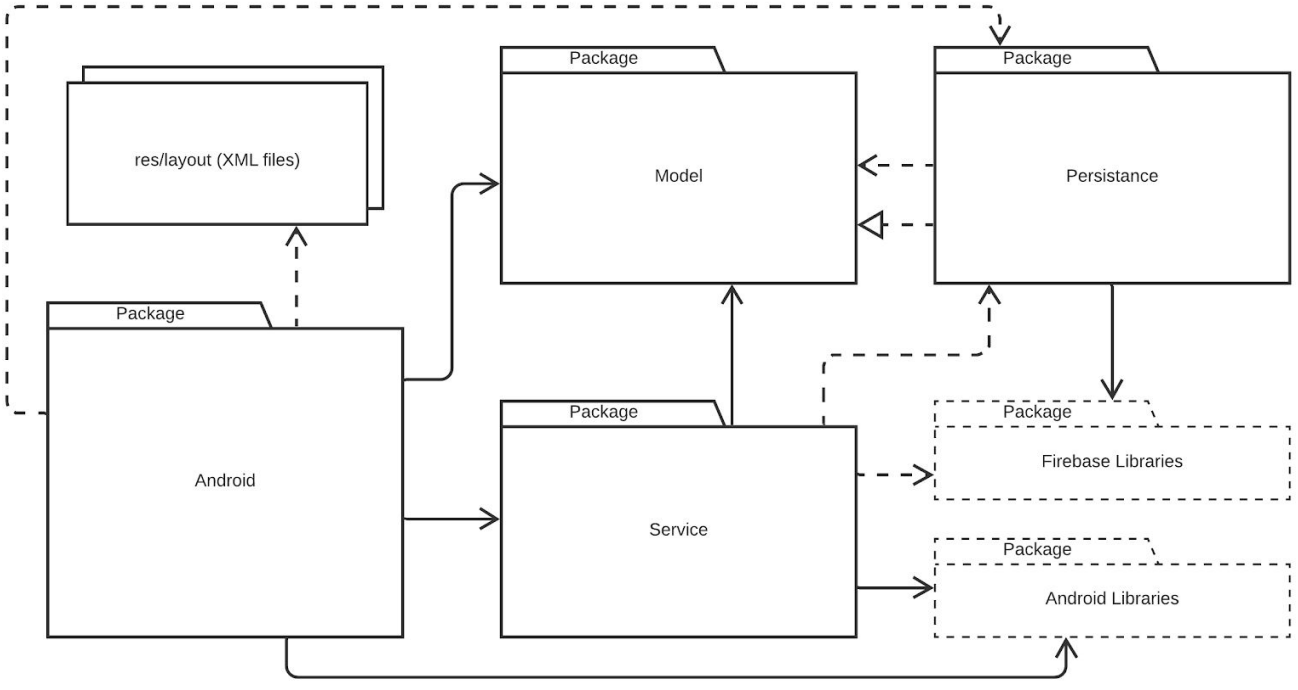

Figure 2. *High-level dependencies between packages.*

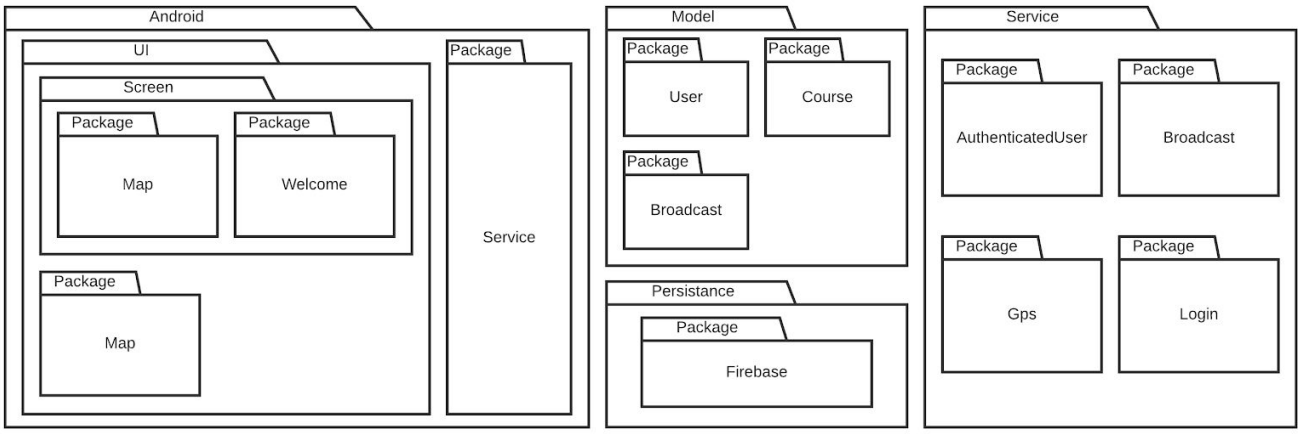

Figure 3. *Package hierarchy.*

3.1 Model package

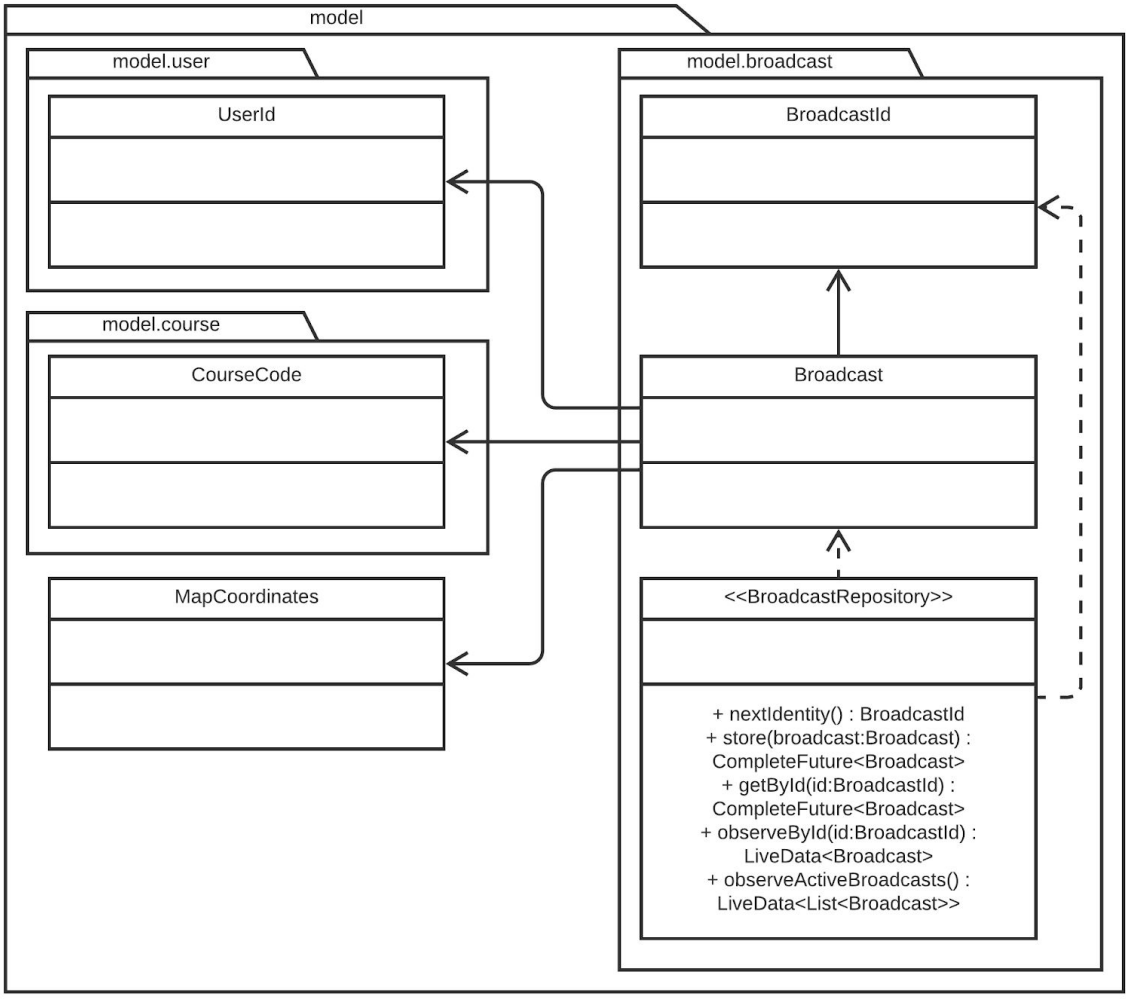

Figure 4. *Dependencies between classes in the model package.*

The model package provides functionality for the broadcast object and the objects a broadcast is built on. This package has no dependencies on the other packages in the program, but all of the other packages are more or less dependent on this package.

3.2 Service package

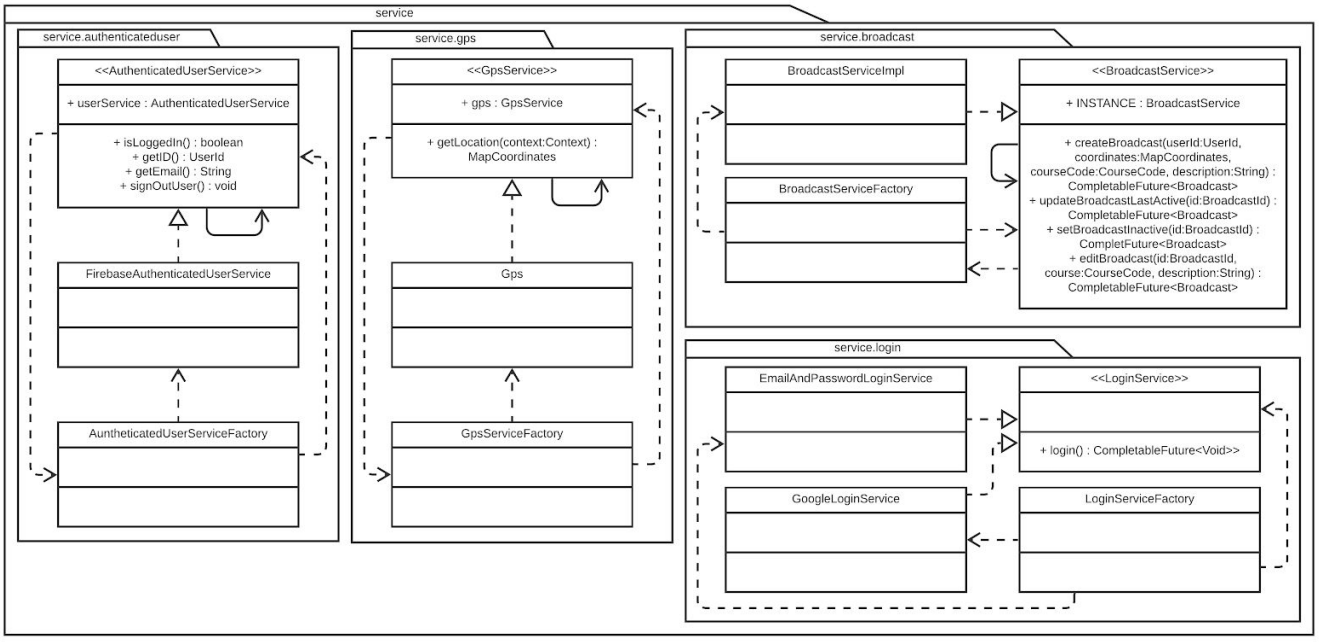

Figure 5. *Dependencies between classes in the service package.*

The service package provides functionality to the Android package which does not fit in the model package. This is because the services have dependencies outside of the application like the Android operating system or the Firebase database. Some services handle objects from the model package. All services utilize interfaces and factories to add abstractions between the implementation of the services and the classes in the Android package that depend on them. The GPS entity from the

3.3 Persistence package

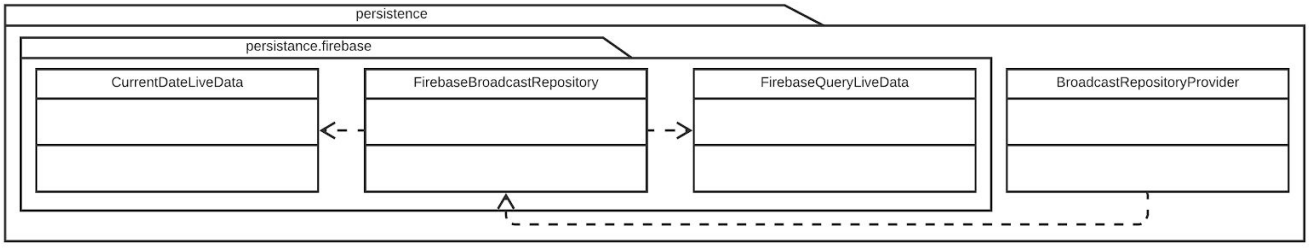

Figure 6. *Dependencies between classes in the persistence package.*

Responsible for communicating between and storing information from the application into the Firebase database. The FirebaseBroadcastRepository is an implementation, backed by Firebase, of BroadcastRepository.

3.4 Android package

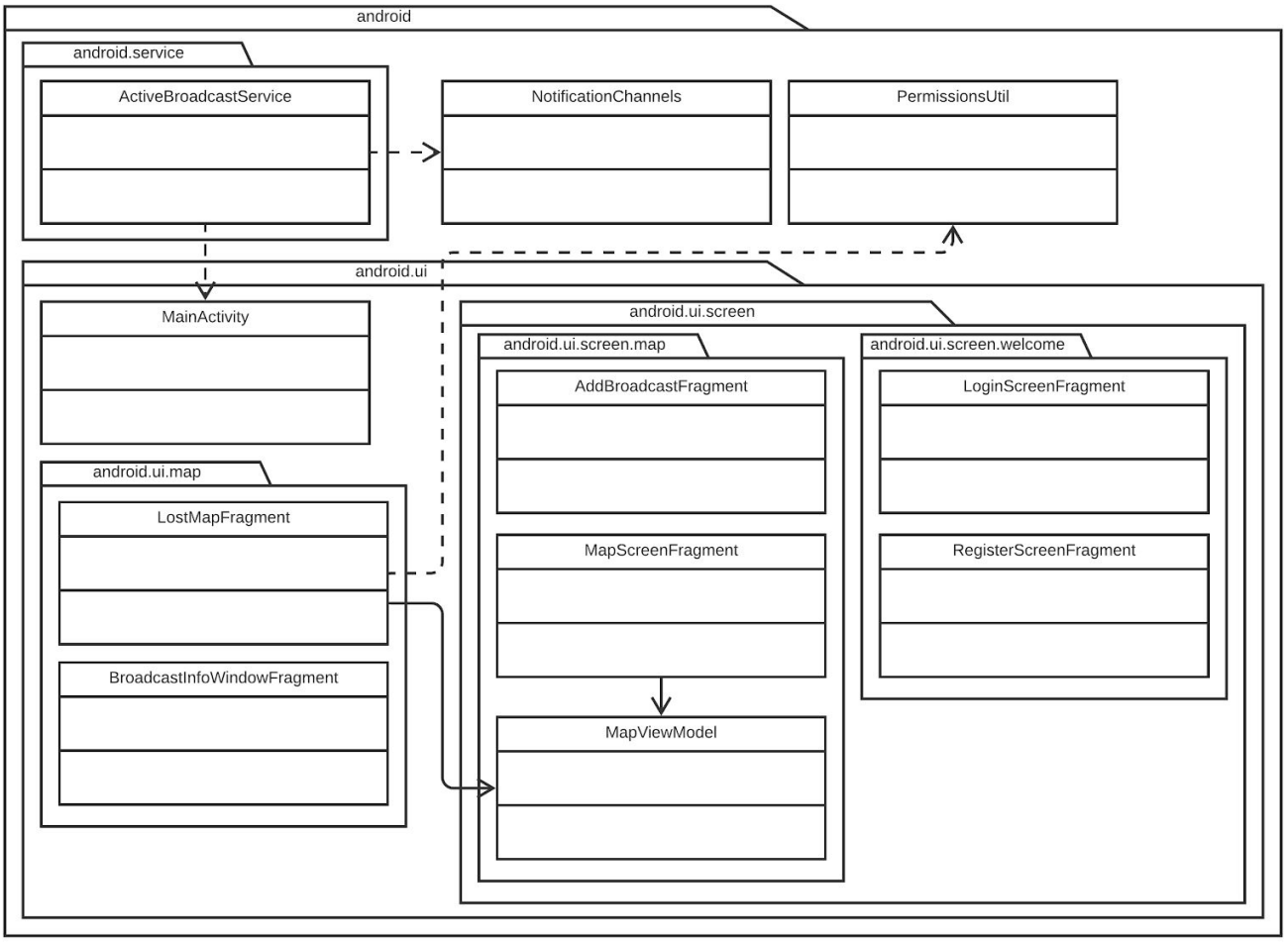

Figure 7. *Dependencies between classes in the android package.*

This package is the one that brings the pieces of the application together, by using the other packages like model and service, with help of the Android system. This package also has responsibility for all interactions between the application and the user. The Android package together with XML files used for layout function as the view of the program.

3.5 Domain model to design model mapping

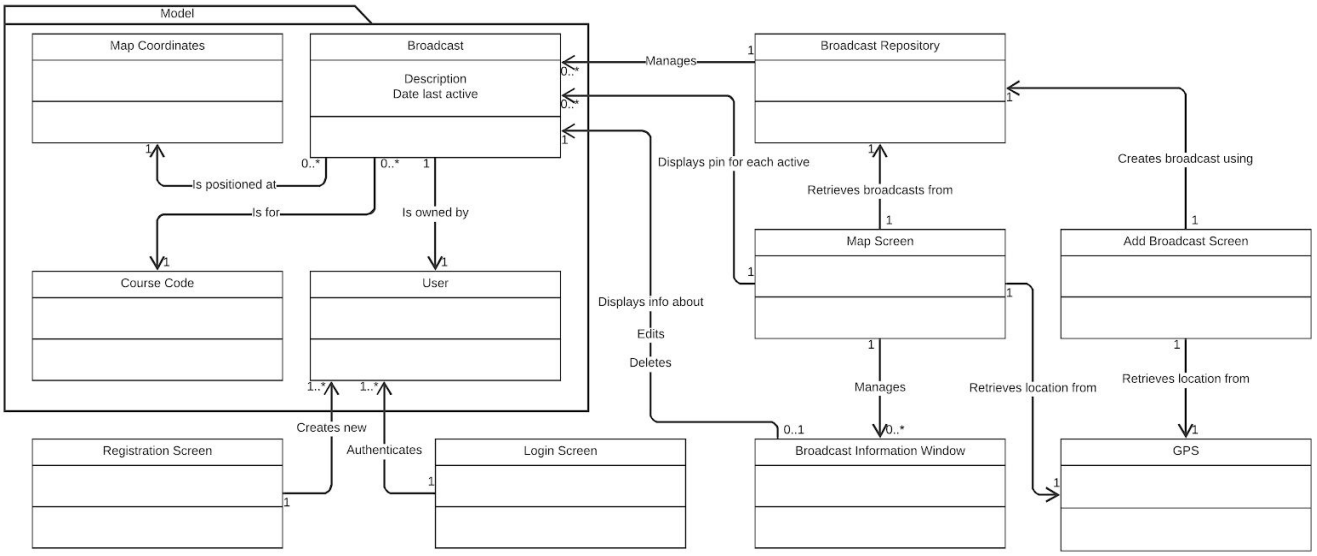

Figure 8. *Domain model UML diagram.*

The model package of the design model corresponds nearly 1-to-1 with the model package in the domain model (Figure 8), with the exception that the User is merely represented by and referred to using a UserId. It additionally contains the Broadcast Repository in the form of an interface.

Broadcast Repository's interface is implemented using Firebase as a backend in FirebaseBroadcastRepository in the persistence package. It provides methods for saving broadcasts to and retrieving broadcasts from Firebase.

The GPS from the domain model is implemented in the form of a service in the service package. This service provides the device's location using a "LocationManager" which is an integrated class in Android.

All screens in the domain model are implemented as "fragments" in the android package that take care of interactions from the user, and make changes thereafter, working together with corresponding XMLs that create the layout and buttons.

Map screen is implemented with the class MapScreenFragment for layout and button functionality, working together with LostMapFragment that provides the map, and map functionality with help of GpsService, and a BroadcastService located in the service package.

Broadcast information window consists of a BroadcastInfoWindowFragment that opens a little window for a broadcast shown on the Map screen. This window provides some information about the broadcast with help from BroadcastService, in order to edit and delete, in LostmapFragment. It also utilizes an AuthenticatedUserInfoService, to get the user id to the currently logged in user, so a user can edit or delete only broadcasts they self have created.

Login Screen is implemented as LoginScreenFragment, with a LoginService located in the service package, that supports both Google accounts and email/password, and takes care of the functionality in the login process. The AuthenticatedUserService is used to provide information about the currently logged in user to other parts of the program.

Registration screen which is implemented as RegistrationScreenFragment, and lets a user register an account in the application, where the account details will be saved in Firebase.

3.6 Implementation of MVC and other patterns

In our application we have implemented the MVC pattern as a MVVM (Model-View-Viewmodel) which is a common implementation of MVC in Android applications. The model package serves as the model in the design pattern. XML and Fragment-classes are responsible for the view, where the XML files contain the layout for the view, and the fragments handle inputs from the user, and adapts thereafter. The view-model consists of interfaces representing different services used by the view, and contains logic the model should not be able to handle by itself, and is not included in the responsibility of the view. The view is dependent on view-model, and model. On the other hand, the model is completely unaware of the view-model, and view, while the view-model is also aware of the model.

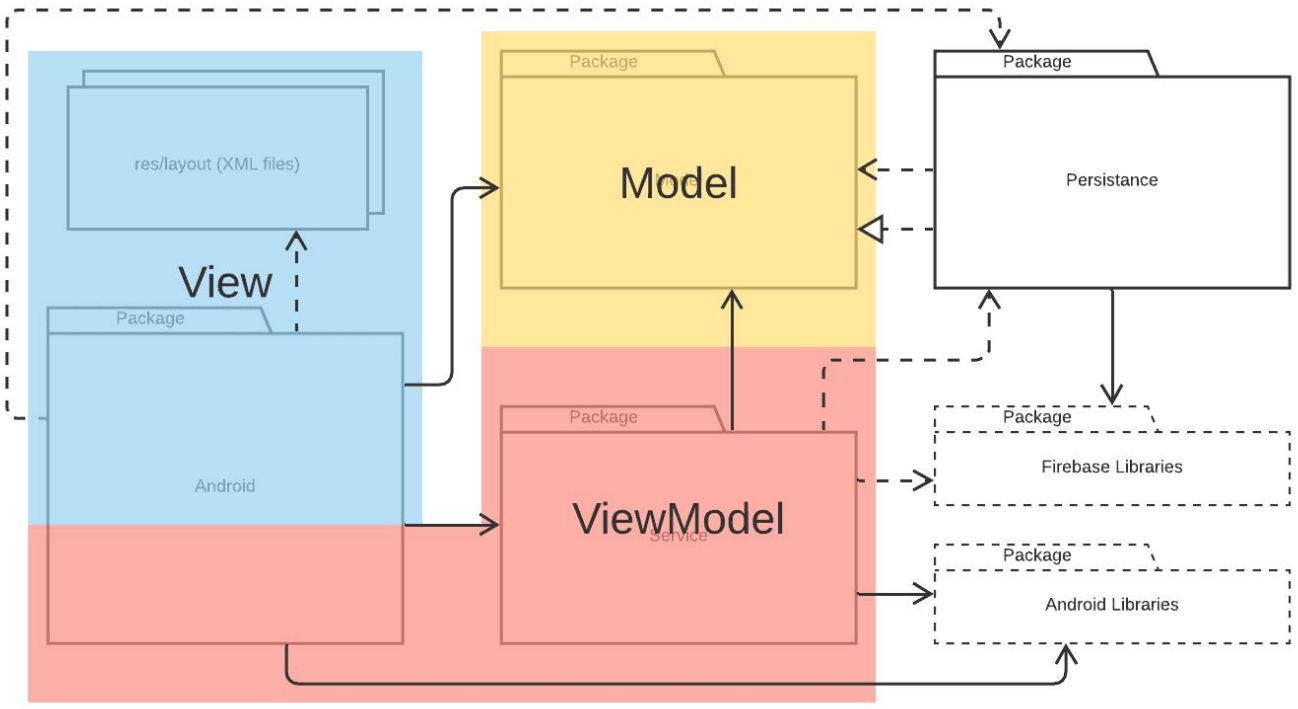

Figure 9. *High-level dependencies between packages (MVVM parts highlighted).*

3.6.1 Factory pattern

We utilize factory patterns for all of our services in the service package to add abstraction from which exact type an object of an interface has. We use factories for GpsService, LoginService, BroadcastService, AuthenticatedUserService and BroadcastRepository.

3.6.2 Singleton pattern

We use multiple singletons patterns in our code where most is the service package, to make services easily accessible to the parts of the program that need them. There is also no need for multiple instances of these services. We use singleton patterns in AuthenticatedUserService, BroadcastService, GpsService and BroadcastRepositoryProvider.

3.6.3 Observer pattern

Since our app needs continuous observation of broadcasts on the map. We have used the observable data holder class LiveData, in order to observe the changed broadcasts. If any change was found then this results in updating the view. The classes that use LiveData are LostMapFragment, ActiveBroadcastService and MapViewModel.

3.6.4 Facade pattern

We have designed our services in the service package with the use of facade patterns. We hide the real implementation of the services by letting them implement interfaces that the parts of the application that use the services depend on. Factories are used to create instances of the services to hide the real class type, and only expose the interface type of services outside the package. We use facade patterns on the packages: login, gps, broadcast and authenticateduser in service package.

3.6.5 Strategy pattern

We use a strategy pattern in LoginScreenFragment, where the user's choice of authentication method will be selected as a strategy. This strategy decides which LoginService is needed for the chosen authentication method, and decides how a part of the login method works.

3.7 Sequence diagrams

In the sequence diagrams the navController is implemented using the Jetpack Navigation framework used for navigating between the different UI screens. The Display is a representation of the loaded UI, i.e., what the user actually sees. Firebase is the cloud based database used for storing information.

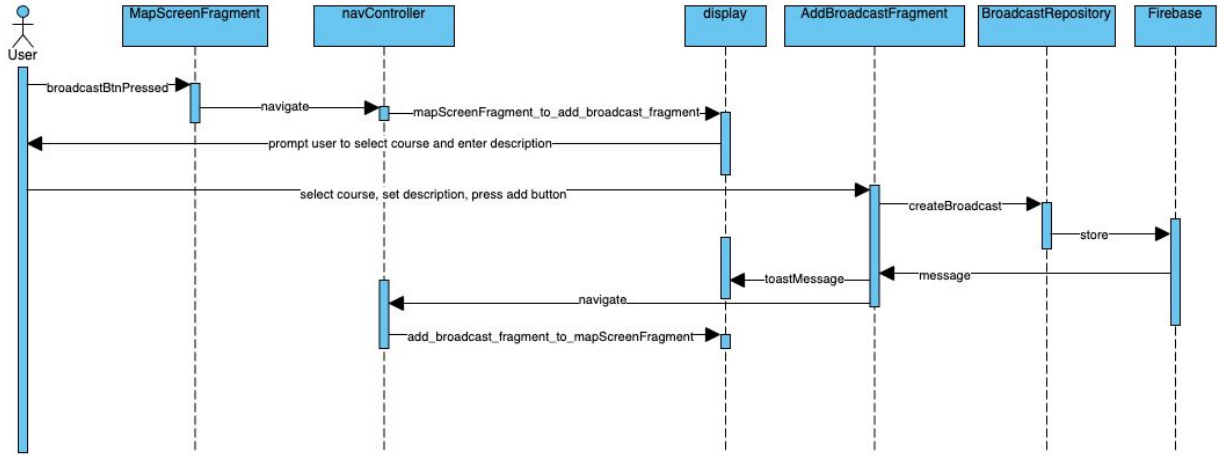

Figure 10. *Sequence diagram for creating broadcast.*

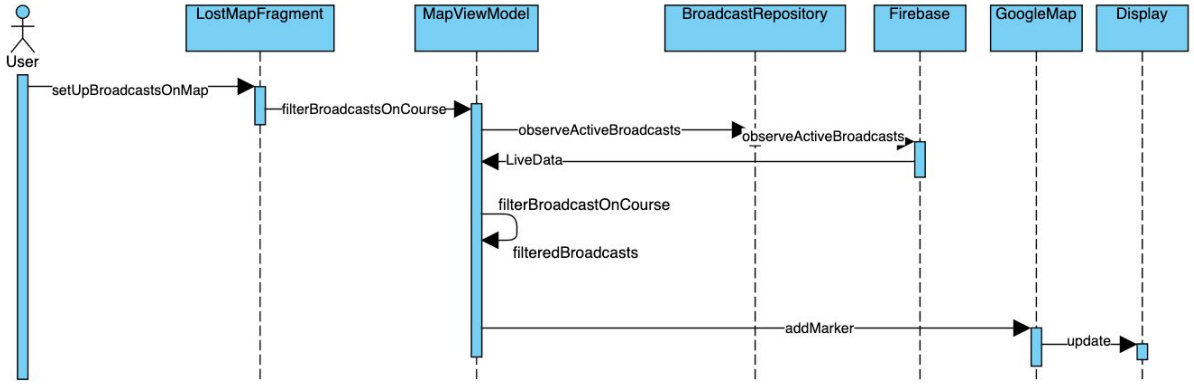

Figure 11. *Sequence diagram for filtering broadcasts on map.*

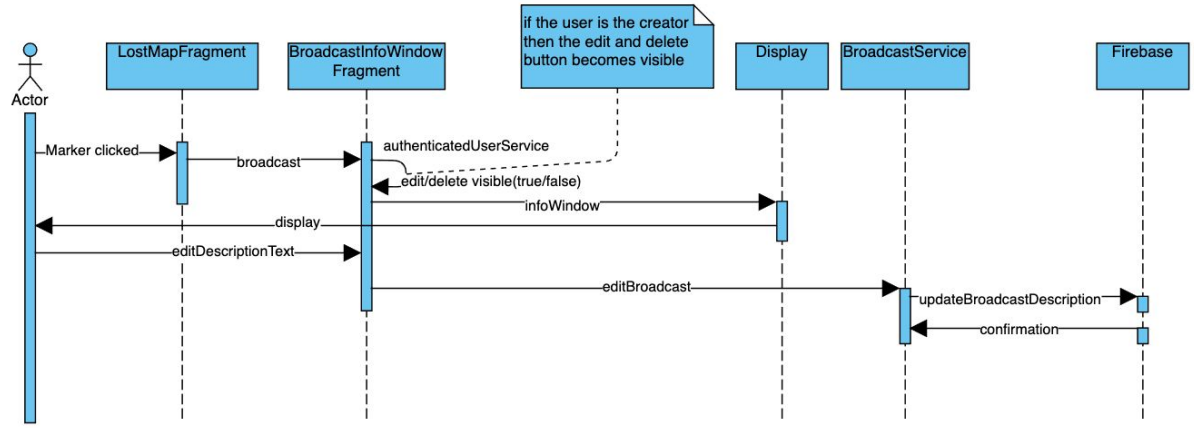

Figure 12. *Sequence diagram for editing broadcast.*

4 Persistent data management

Our application uses the Firebase realtime database for saving user's data, like broadcasts. Additionally, we use Firebase authentication to store users when making an account. The realtime Firebase database enables our application to store the coordinates of a broadcast, last time active, course code, description, when the broadcast is created, and which user created it. FirebaseBroadcastRepository is the implementation used for the interface BroadcastRepository. The implementation connects to Firebase and does the work of translating the format stored in the database to objects in the model, and vice versa.

Local resources like images, view layouts (XML), styling (XML), configuration (XML) and translation strings (XML) all follow the standard Android/Maven project structure where they are located in the directory "src/main/res/". Most of these are automatically loaded and used by the Android system. Whenever they are referenced in our application's code it is done through auto-generated classes like "R" (resource) and, e.g. FragmentMapScreenBinding (corresponding to: src/main/res/layout/fragment_map_screen.xml).

5 Quality

5.1 Testing

Our tests consist of multiple levels: unit, integration, UI tests for isolated fragments and UI tests for high-level interaction with multiple components. Before our tests were mostly unit tests but we have diligently been working on our instrumental tests and now that is what it consists most of. All our tests are created using JUnit and additionally our UI tests utilize Espresso from the Android Test library. For continuous integration we have used GitHub Actions which builds the project and runs tests on each push (https://github.com/DIT212-Alpha/DIT212-Alpha/actions).

To make it easier to find our tests we recommend you to switch to Android view instead of Project view in the Project panel of Android Studio. Instrumental tests are in the "(androidTest)" package while the unit tests are in the "(test)" package. We have added "JaCoCo Java Code Coverage Library" which let us know how much testing coverage we have and where we have it. We recommend using this when testing our application. Even though the application does not have full test coverage, it has been tested and covered for the most part, which is more than 90% statement coverage, and certainly has covered the most important parts.

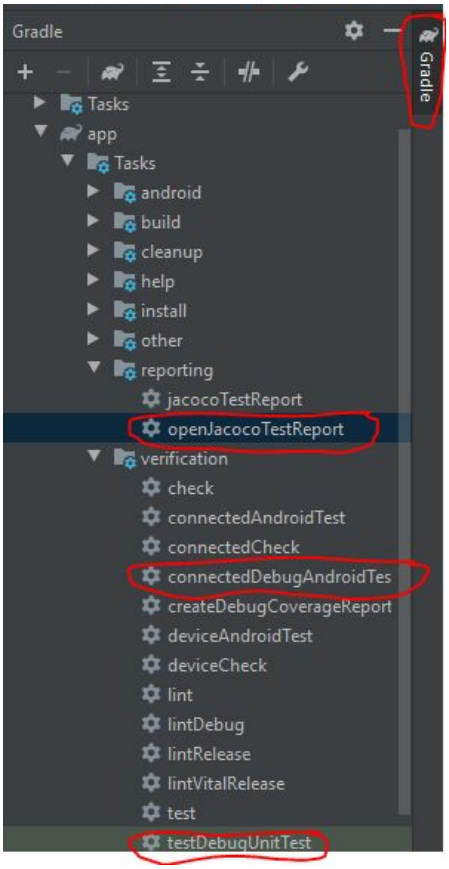

Figure 13. *Tests and JaCoCo usage in Android Studio.*

As shown in Figure 13 to run all tests and open the coverage report, you should:

1. Click on "Gradle" on the right side of Android Studio.
2. Run the Unit tests by clicking on "testDebugUnitTest".
3. Run the Instrumental tests by clicking on "connectedDebugAndroidTest" (make sure your AVD is on and restart the AVD again if it is slow).
4. Click on "openJacocoTestReport" and you should be able to see the following report (Figure 14).

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
cse.dit012.lost.service.login		42 %		33 %	6	11	23	37	4	8	1	3
cse.dit012.lost.android.service		82 %		62 %	8	21	15	83	3	13	1	2
cse.dit012.lost.android.ui.screen.welcome		91 %		95 %	5	43	10	117	4	31	0	2
cse.dit012.lost.persistance.firebase		94 %		78 %	5	35	5	94	2	28	0	7
cse.dit012.lost.android.ui.screen.map		93 %		81 %	7	31	7	79	4	23	0	4
cse.dit012.lost.android.ui.map		97 %		66 %	4	28	5	104	1	22	0	2
cse.dit012.lost.model.broadcast		98 %		77 %	4	30	0	51	0	21	0	2
cse.dit012.lost.model		98 %		70 %	3	12	0	26	0	7	0	1
cse.dit012.lost.model.user		96 %		75 %	2	8	0	10	0	4	0	1
cse.dit012.lost.model.course		96 %		75 %	2	8	0	10	0	4	0	1
cse.dit012.lost.android		97 %		50 %	2	4	0	8	0	2	0	2
cse.dit012.lost.service.broadcast		100 %		n/a	0	10	0	18	0	10	0	3
cse.dit012.lost.service.gps		100 %		100 %	0	6	0	10	0	4	0	3
cse.dit012.lost.service.authenticateduser		100 %		100 %	0	8	0	10	0	7	0	3
cse.dit012.lost.android.ui		100 %		n/a	0	2	0	5	0	2	0	1
cse.dit012.lost.persistance		100 %		n/a	0	2	0	2	0	2	0	1
Total	256 of 3 057	91 %	34 of 142	76 %	48	259	65	664	18	188	2	38

Figure 14. *JaCoCo coverage report.*

5.2 Structural analysis

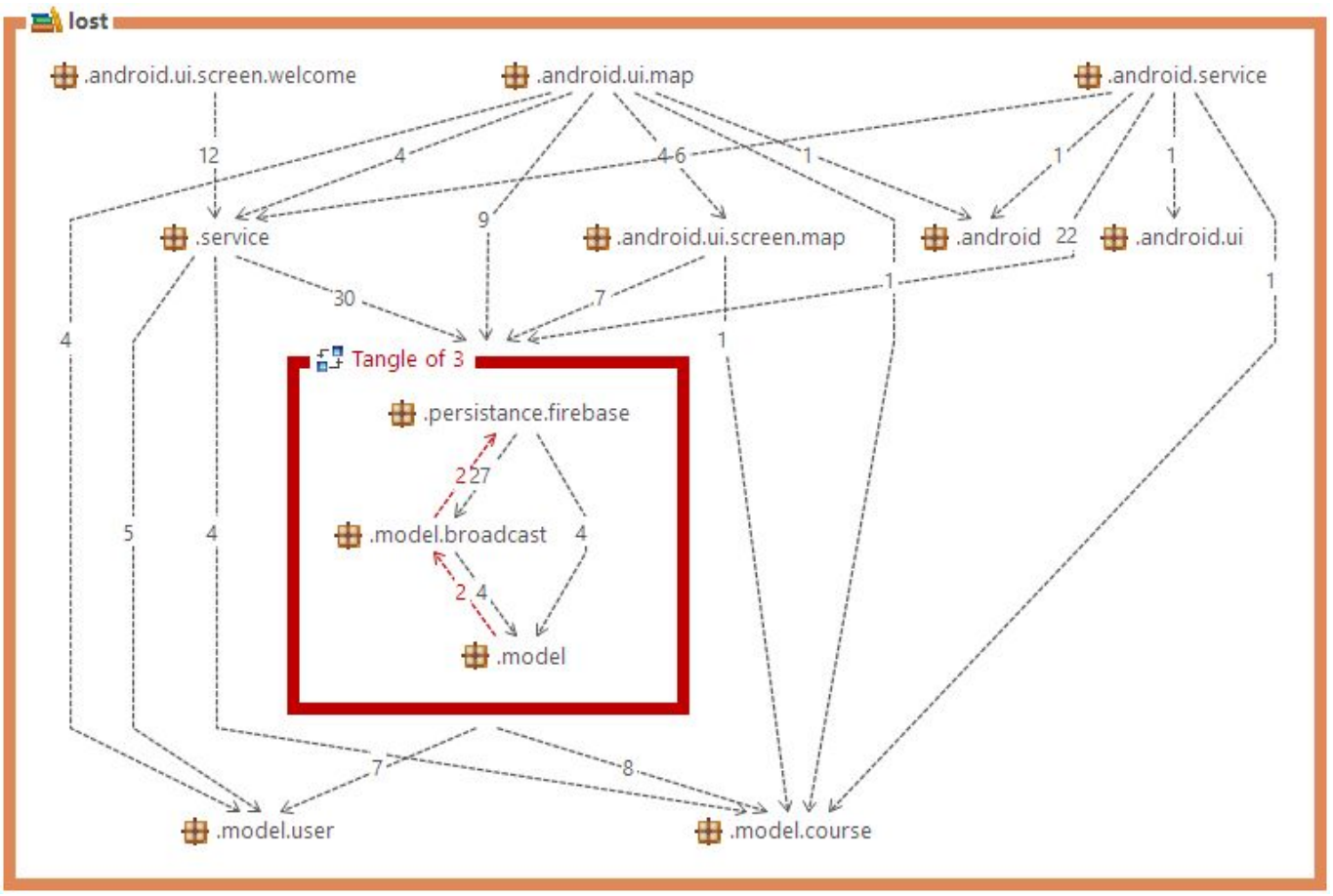

Figure 15. *STAN dependency graph before fixing cycles.*

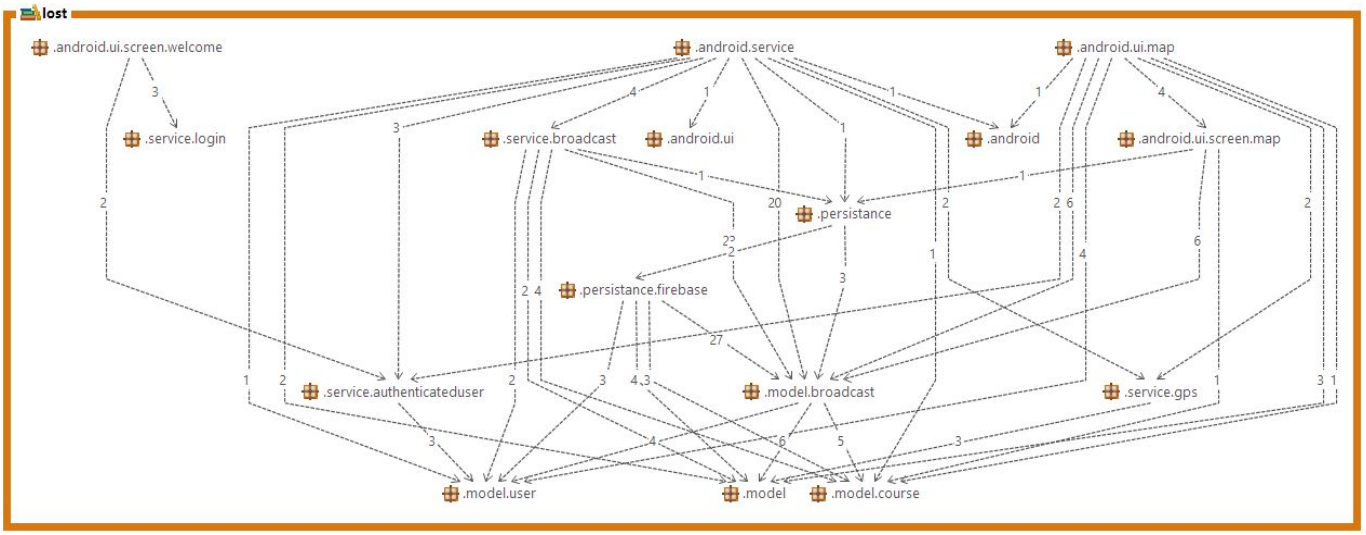

Figure 16. *STAN dependency graph of final version.*

To analyze the structural quality of the project's code we used the tool STAN (Structure Analysis for Java). This allowed us to find and solve a couple of cycles in the code (Figure 15), which resulted in the structure seen in Figure 16.

5.3 Remaining known issues

We currently have no major issues with regards to our completed user stories.

Regarding testing:

- When testing the Android Tests then it might be necessary to dismiss the popups manually else it could mess with the tests.
- The tests seem to depend on each other's different states, which means that the tests work when testing on their own but when testing the whole test package it can affect the other tests.
- It is very likely that you have to try to start the tests a few times in the beginning before testing the tests properly due to the AVD lagging and loading in the beginning.
- The reason for not having full coverage is mostly because it is hard to test the Google Sign in because of Google's Authentication you have to go through before testing.
- Furthermore we can not get full coverage on:
    - LostMapFragment.onPermissionRequestResult(boolean), it is hard to test a third-party interface we cannot control.
    - LostMapFragment.requestGeolocationPermissions(), it is hard to test a third-party interface we cannot control.
    - ActiveBroadcastService.NotificatioinActionsReceiver.onReceive(Content, Intent), it is hard to test a third-party interface we cannot control.
    - ActiveBroadcastService, it was hard to assert that the service was performing what it was intended to do since it is run by the Android system, out of our control to some degree.

5.4 Access control and security

In our application, the user goes through a login and a registration screen to gain access to the rest of the application. The application checks if the user already has an account or if the user wants to register an account when starting the application in order to login and use the application. Login details are saved in our authentication provider Firebase, after completing the registration process. Either email and password or a Google account is possible to use as credentials.

When a broadcast is created, the user id from the user who is currently logged in on the device that creates the broadcasts, is saved together with the broadcast in Firebase. When an InfoWindowFragment is created for a broadcast retrieved from Firebase, it compares the user id from the creator of the broadcast with the user id of the user who is logged in on the device displaying the fragment. This ensures that a user can only edit or delete broadcasts they have created themselves, and are not able to do so on broadcasts created by other users.

The Firebase realtime database allows for adding security and validation rules. We do not use the security rules at all, since it was not a major priority, meaning that any user can technically read and write to any part of the database. However, we do use validation rules

that make sure to keep the schema of the database intact. It checks both the structure and the types of the database fields to make sure invalid data cannot be stored.

6 References

[1] *Android Studio and SDK tools*. Google, JetBrains, 2020.
[2] J. Tamplin and A. Lee, *Firebase*. Google, Alphabet.
[3] "Navigation," *Android Developers*. https://developer.android.com/guide/navigation (accessed Oct. 01, 2020).
[4] "Maps SDK for Android," *Google Developers*. https://developers.google.com/maps/documentation/android-sdk/overview (accessed Oct. 01, 2020).
[5] K. Beck, E. Gamma, D. Saff, and K. Vasudevan, *JUnit 5*. 2020.
[6] Android developers, "Espresso," *Espresso*. https://developer.android.com/training/testing/espresso (accessed Oct. 21, 2020).
[7] H. Dockter *et al.*, *Gradle Build Tool*. 2020.
[8] L. Torvalds, *Git*. .
[9] L. Torvalds, "Build software better, together," *GitHub*. https://github.com (accessed Oct. 01, 2020).
[10] EclEmma, "JaCoCo Java Code Coverage." https://www.eclemma.org/jacoco/.
[11] Bugan IT Consulting, "STAN, Report Generation." http://stan4j.com/reports/ (accessed Oct. 21, 2020).

Platforms:

- Android (https://www.android.com/)
- Firebase (https://firebase.google.com/)
- GitHub (https://github.com/)

External Tools:

- Android Studio (https://developer.android.com/studio)
- Gradle (https://gradle.org/)
- Git (https://git-scm.com/)
- JUnit (https://junit.org/junit4/)
- Espresso (https://developer.android.com/training/testing/espresso)
- JaCoCo (https://www.eclemma.org/jacoco/)
- STAN (http://stan4j.com/)

Libraries:

- Android SDK (Bundled with Android Studio)
- Jetpack AndroidX Support Libraries (https://developer.android.com/jetpack/androidx)
- Jetpack Navigation Graph (https://developer.android.com/guide/navigation)
- Jetpack Architecture Components (https://developer.android.com/topic/libraries/architecture)
- Google Play Services

    - Maps (https://developers.google.com/maps/documentation/android-sdk)
    - Auth (https://developers.google.com/identity/sign-in/android)
- InteractiveInfoWindowAndroid (https://github.com/Appolica/InteractiveInfoWindowAndroid)
- Firebase (https://github.com/firebase/firebase-android-sdk)
    - Realtime Database
    - Authentication
    - Analytics