Analysis and Design Document

Student: Zavaczki Péter - Tibor
Group: 30433

Revision History

Date	Version	Description	Author
01/Apr/18	1.0	Beginning document	Zavaczki Péter - Tibor
15/Apr/18	1.1	Revision 1	Zavaczki Péter - Tibor

Table of Contents

I. Project Specification

ITA is an Android based application for traffic participants to cooperate by sending traffic alerts to a central database, from where other users can see them in real time so that they can adapt their route and/or driving style accordingly.

II. Elaboration – Iteration 1.1

1. Domain Model

The domain model of package delivery is presented with the conceptual class diagram in Illustration 1 below. The Users are modeled to have a username, an email to register with and a password, additionally, every user can submit a traffic alert from the moment of their creation. The alerts have a type (ex.: police, roadworks, accident etc.) and a coordinate location that is queried from the device when the alert is submitted (ex.: 46°46'20.0"N 23°35'07.8"E).

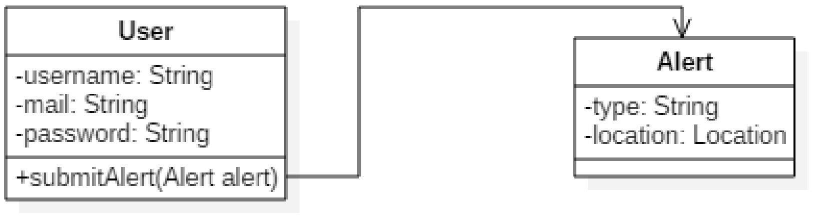


Illustration 1: Domain model conceptual class diagram

2. Architectural Design

2.1 Conceptual Architecture

The application will be implemented based on the client-server architectural pattern, since the users should be able to access the service provided by the system at all time. This pattern also allows the main functionality of the application to work correctly, allowing multiple users to be simultaneously connected, all of them being able to send and receive alerts in real time. The view layer of the application is present on the user side along with a portion of the controllers. The server connects the database, which may be hosted locally or on a different domain, to the local data access layer and a portion of the controllers. The Controller layer is split into two sublayers: the client side controller layer and the server side controller layer. This conceptual architecture can be seen on Illustration 2.

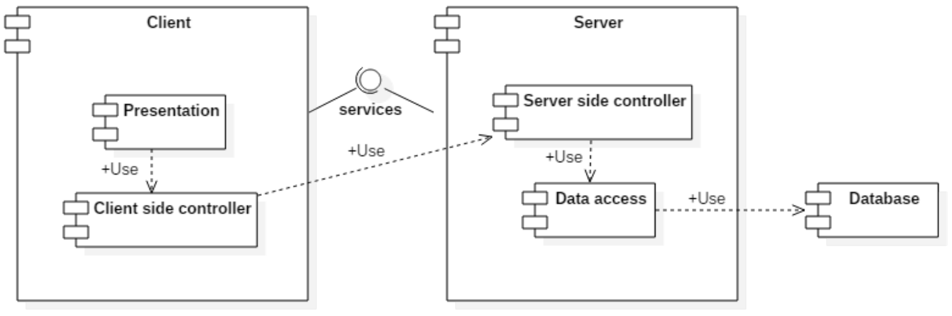


Illustration 2: Conceptual architecture diagram

2.2 Package Design

The package diagram in Illustration 3 represents the structure in which the system's components are organized. Packages encapsulate classes which are semantically connected to a certain extent, as such, for example every data access object will be in the DAOs package.

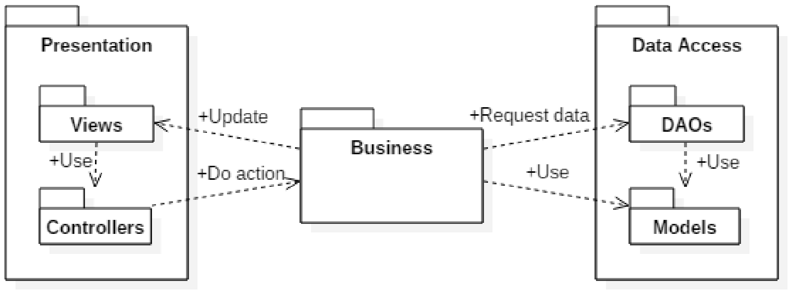


Illustration 3: Conceptal package diagram

2.3 Component and Deployment Diagrams

The project ultimately sums up in two component modules, the visual application, running on the client side device, and the server side logic module. This can be seen on Illustration 4.

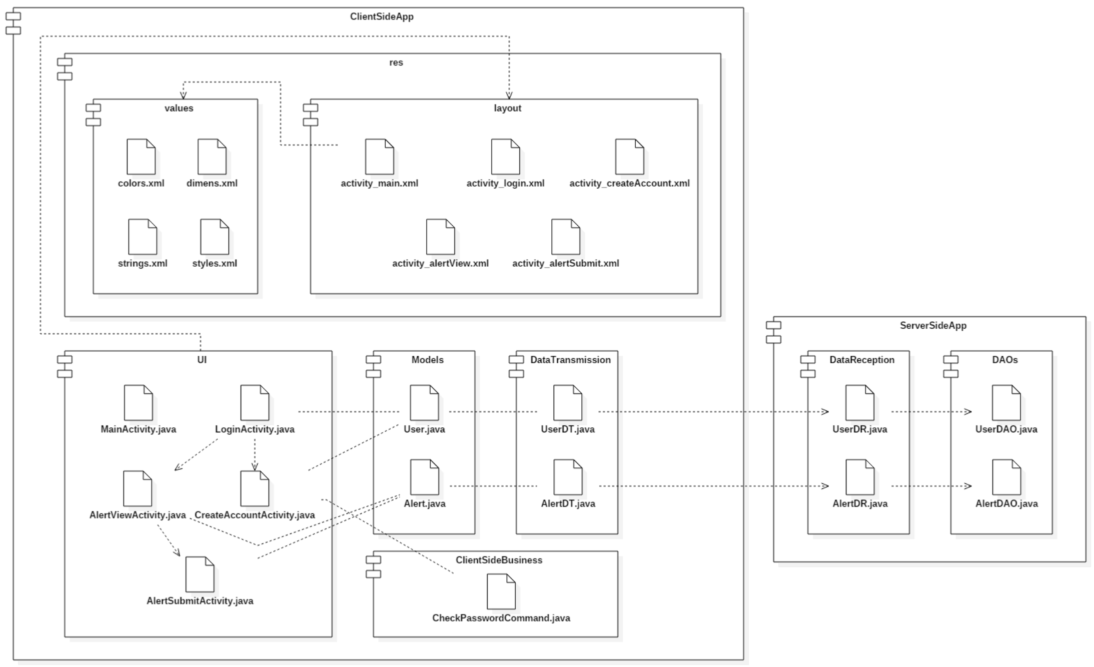


Illustration 4: Conceptual component diagram

The application's deployment consists of deploying the app on the execution environment, which then runs on the client's device. The second part of the deployment is deploying the server app on its own execution environment, so that the clients can connect to it. This can be seen in Illustration 5.

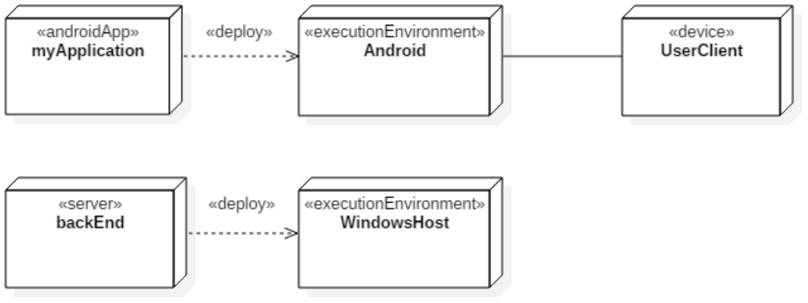


Illustration 5: Conceptual deployment diagram

III. Elaboration – Iteration 1.2

1. Design Model

1.1 Dynamic Behavior

The application has a very simple functionality set: a user can create a new account to access the application, or he can log into an existent account, thus being able to send new alerts and view the current alerts, which is the everyday users' use case.

1.1.1 Create Account

This behavior refers to the first step the user has to do to access the application, creating the account, during which the user should input their preferred username, a valid email address, and a password. After the required information is inputted, the application creates a User type object, which is then sent to the Data Access Object related to the Users and it is inserted into the database, after which the user gets his view switched back to the login view. This behavior can be seen on Illustration 6 and 7.

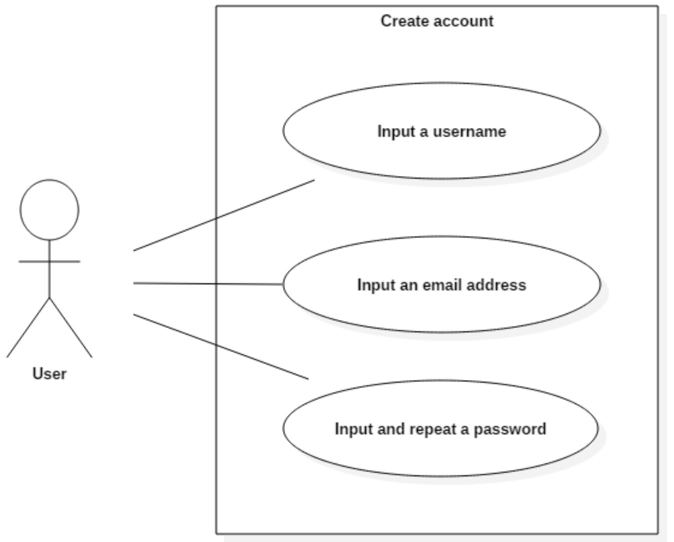


Illustration 6: Create account use case diagram

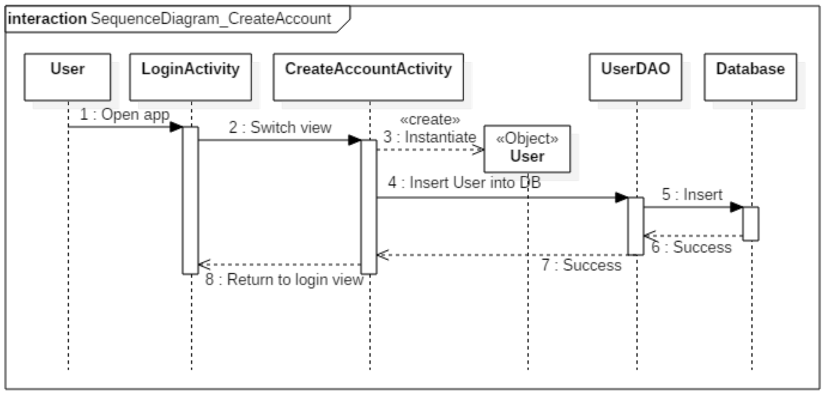
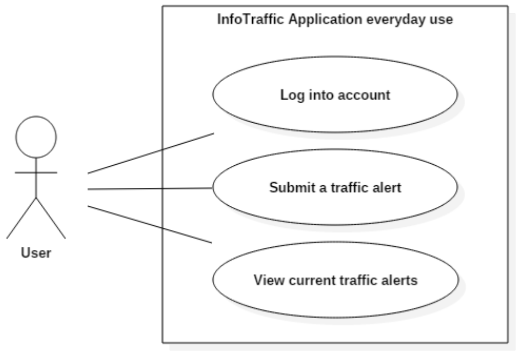


Illustration 7: Create account sequence diagram

1.1.2 Everyday use

This behavior refers to the activities a user can do in their everyday use of the app. After the user logs into his existing account, he is redirected to the UI where he can see the currently submitted alerts. Alternatively, if he notices some traffic incident which is unsubmitted, he can submit it, by switching to the respective UI and submit it. The application then instantiates an Alert type object, sends it to the Data Access Object related to the Alerts and it is inserted into the database. After this process is done, the UI is automatically switched back to the Alert view UI. This behavior can be seen on Illustration 8 and 9.



InfoTraffic Application everyday use
Log into account
Submit a traffic alert
View current traffic alerts
User

Illustration 8: Everyday use use case diagram

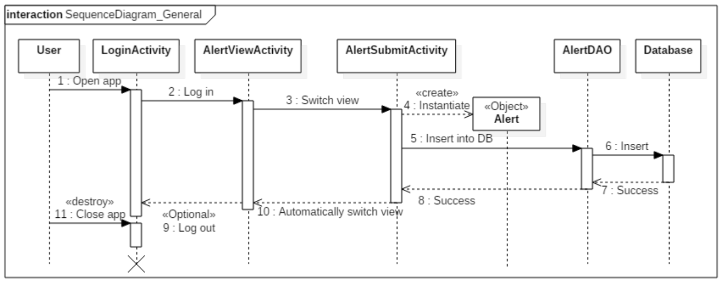


Illustration 9: Everyday use sequence diagram

1.2 Class Design

The most important patterns used in this project's creation are the Command, Observer and Factory. The Command pattern is used to add functionality to the UI's buttons, the Observer pattern is useful for displaying data onto the UI, while the Factory pattern is used to create a connection of the back-end to the database. A conceptual class diagram can be seen on Illustration 10.

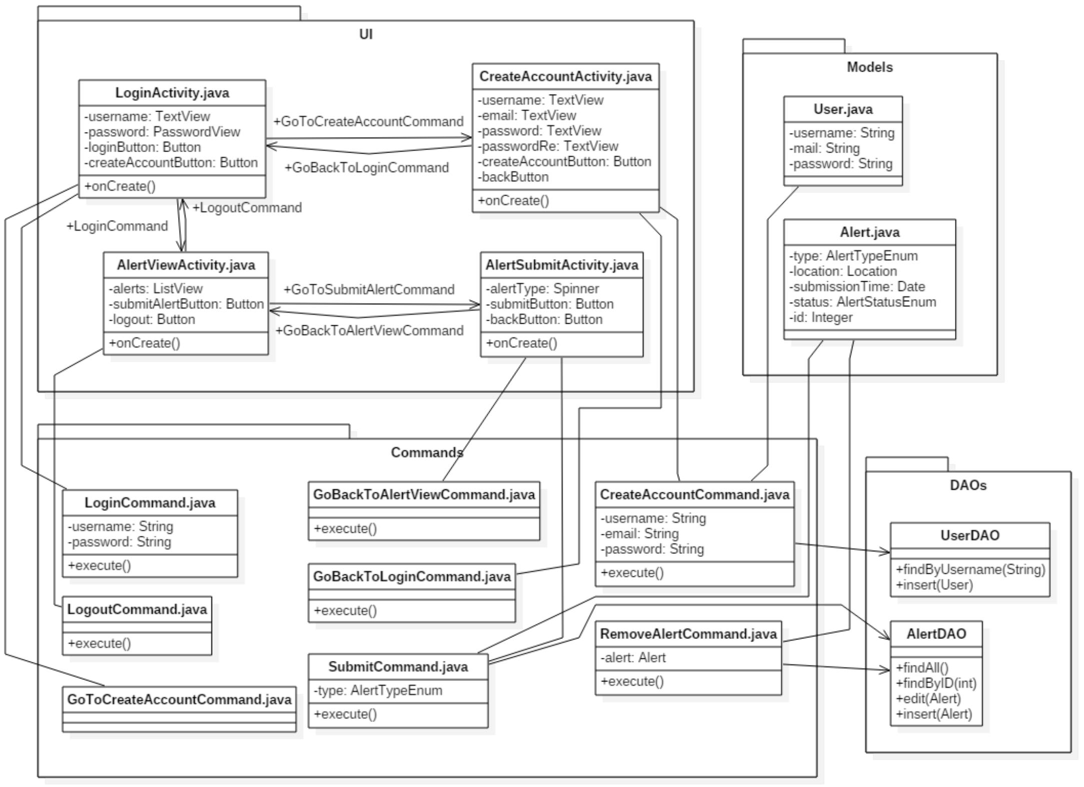


Illustration 10: Conceptual Class Diagram

2. Data Model

The presented system relies on two data models: the User data and the Alert data. The User data encapsulates a user's username, email address and password, while the Alert data encapsulates information about the submitted alerts, such as the alert type, the location where it was submitted, the time when it was submitted or its status (active, deleted). These data models can be seen in Illustration 11 and 12.

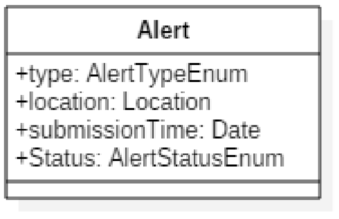


Illustration 11: Conceptual Alert data model

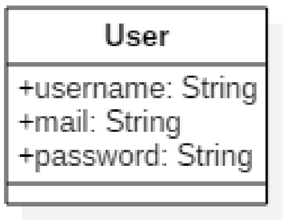


Illustration 12: Conceptual User data model

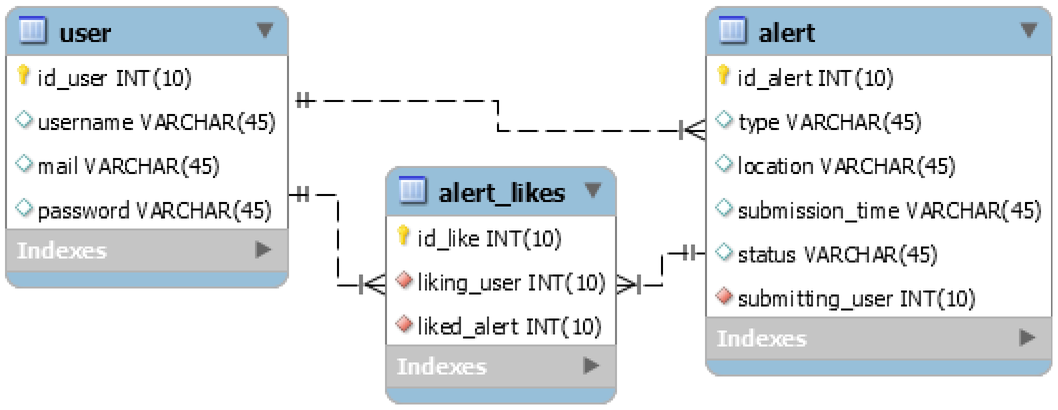


Illustration 13: Database model

3. Unit Testing

The project's unit tests regard the login and user registration.
For the login, the tested element could be the format of the email address: a userame, a '@' sign, an email provider name and a domain.
For the user registration, the tested element can be composed of the login test for testing the email validity. Beside the email validation, an additional password validity test, where we can impose a format (Capital letter, small letter, number, symbol, minimal length) and beside that, the two passwords inputted at the registration screen should be equal.

IV. Elaboration – Iteration 2

1. Architectural Design Refinement

The architecture is according to the planned: client-server with a three layer architecture. The three layers are the UI, the business and the DAO layer. The UI layer is based on an MVC architecture. The Model is common on the client and server side, it being composed of the Alert, AlertList, User, Like classes. The View and Controller are exclusively found on the client side. They always come in pairs: an xml layout (view) file with a java activity (controller) file. The business layer is common between the client and the server. Transmitting the data between the client and the server is done by sending strings composed of commands and information, for example, for the list of alerts, the server sends an AlertList object serialized using gson to the client, then the client deserializes it and assigns it to the ListView on the Android layout. The data access layer can be found only on the server side and it is implemented using the jdbc API for MySQL, subsequently the database management system for storing the necessary data is also MySQL.

2. Design Model Refinement

The class diagram is more or less the same, it shows the architectural composition of the app: three tier, with the UI being an MVC and exclusively on the client, the DAO being exclusively on the server and the model from the MVC and the business layer being common.

V. Construction and Transition

1. System Testing

As of writing this document there are no unit tests implemented.

2. Future improvements

Future improvements can include: unit tests; converting the client connection to something else than an

AsyncTask, which is supposed to be a -short- background task, thus enabling the full functionality of every section of the app, without creating a new client connection for every task to be performed; enlarging the set of commands to cover the full proposed functionality set of the application; modifying the UI layer to work with the full power of the Android platform (fragments); converting the DAO API to Hibernate (eventually, not really but meh).

VI. Bibliography

https://stackoverflow.com/*
https://developer.android.com/*
https://github.com/buzea/SoftwareDesign2018
https://martinfowler.com/eaaCatalog/transactionScript.html