System Design Document for PlankaGBG

Lucas Karlsson, Joakim Ohlsson, Seif Bourogaa,
Joakim Tubring, Filip Hanberg

2020-09-25
version 1.1

1 Introduction

This document contains technical information about an application that documents ticket controllers position in a tram network. The application is in this moment built as an Android application. Which displays the information and will allow for search queries on user reports of ticket controllers activity and allows the user to create new reports on their activity.

1.1 Definitions, acronyms, and abbreviations

1. **GUI**: The graphical interface, how the functionality is displayed to the user.
2. **Model**: The package of the program that will manage all functionality of the application.
3. **View**: The package of program that contains the entire GUI.
4. **Controller**: The package of the program that connects the View package to the Model package.
5. **RunTime**: The period where the program is running.
6. **J-Unit**:Simple framework that lets the programmer write repeatable tests.

2 System architecture

2.1 Overall description

At the moment all application components are run locally. There are plans on implementing a Server-Client model using a web-API to connect the server side with the client. The Server side will contain an server application and a database relaying information to the client-side.

2.2 Classes

2.2.1 AbstractReport

Responsible for creating Report objects. Contains the number of controllers, timestamp of report and and possibly an image.

2.2.2 Graph

The graph representation, contains an adjacency list containing nodes and their connections to other nodes. Also responsible for creating, maintaining, traversing and updating the graph structure. The Graph class is dependent on the Node class for creating nodes. The class is also dependent on Station and Route classes for creating routes and stations.

2.2.3 Route

A representation of a route. Contains a line number and a list of stops on the route. Dependent on Node class for creating nodes.

2.2.4 Station

A representation of a station in the tram network contains the name of a station and a list of all individual stop positions. Dependent on Node class for creating nodes.

2.2.5 Incident

A representation of an incident. Contains a list of reports connected to the incident and the collective trust factor of the incident. Dependent on AbstractReport for the list of reports.

2.2.6 ReportRoute

A representation of an report bound to a station. Contains an AbstractReport and a Route.

2.2.7 ReportStation

A representation of an report bound to a route. Contains an AbstractReport and a Station.

2.2.8 Model

The model class will be the connection between the model and the GUI. At the moment it has not yet been implemented, more information on this later.

2.3 Communication protocols

At the moment there are no communication protocols in use as the application is only run locally.

2.4 Description of program flow

2.4.1 Startup

1. Objects are initialized.
2. GUI is created.
3. Network object reads in files containing information about every tram line
4. Network then create Routes that should be in the network and adds them.
5. Network create Stations and adds every possible state a Station has.
6. Network will hold and manipulate any and all data.
7.

2.4.2 RunTime

We have not yet implemented the frontend, so there is currently no RunTime. **Note** Add

2.4.3 Closing

There are no methods that write data to offline storage, once the application is closed the data is simply deleted. Any relevant data is in the application.

3 System design

We have decided to use Model-View-Controller. This is what our Domain Model looks like:

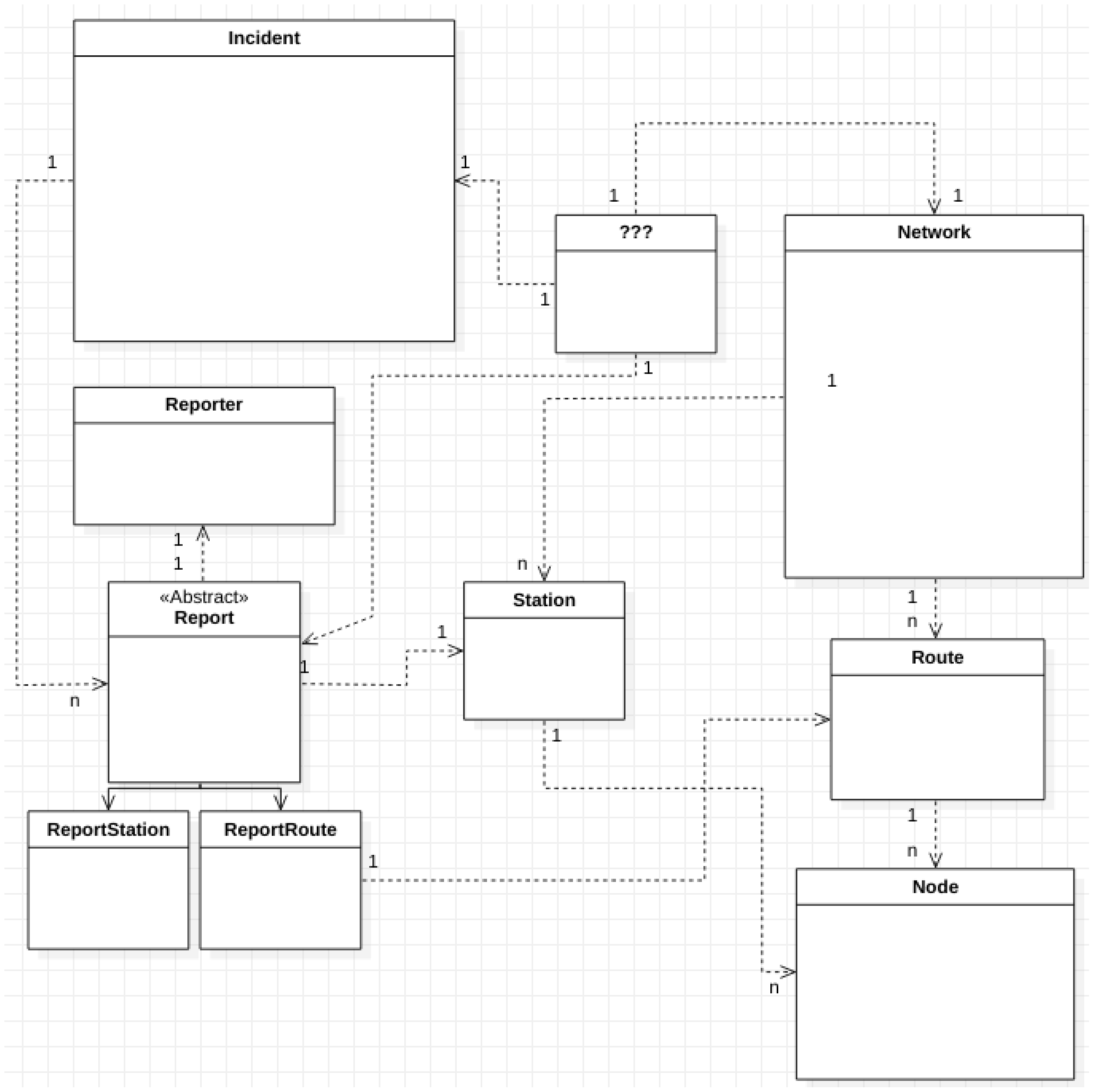

Figure 1: Domain Model

In the current iteration the focus has been on the Model Part of our MVC as shown above. Most of the classes has been implemented apart from the Model class that binds everything together (this should be done this week).

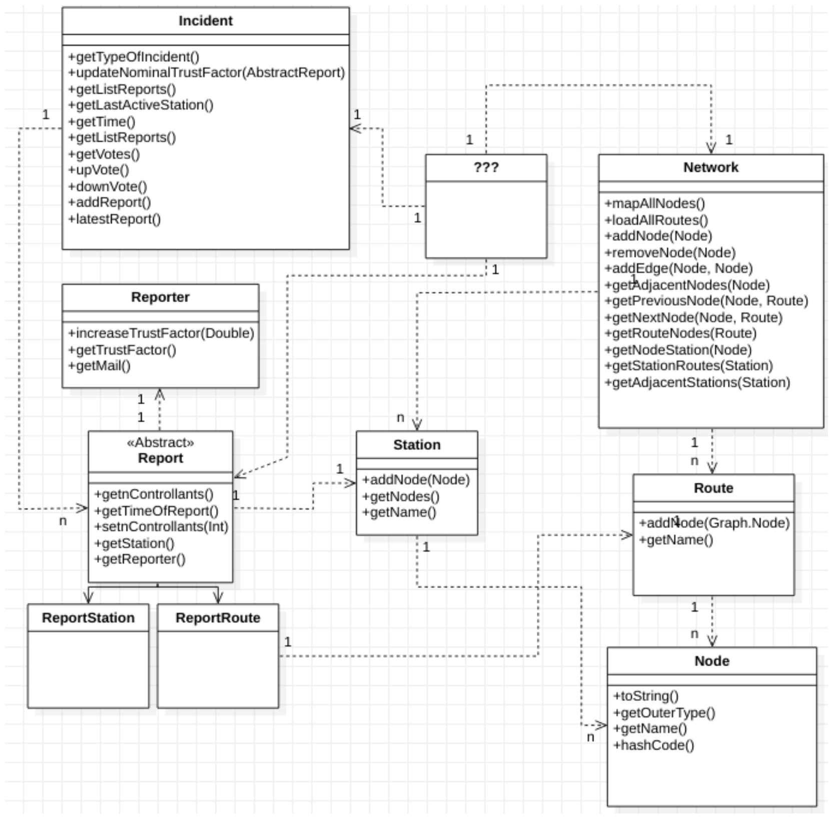

Figure 2: Design Model

We intend connect View to Model using Controllers for input and Observer Pattern for output.

4 Persistent data management

There is no persistent data apart from the .txt-files used to create the Network. These are stored in the application, under ”Assets”.

5 Quality

Quality assurance will consist of tests and code reviews. Testing is done using JUnit-tests. Tests will be provided in test-directory and the code will be tested continuously throughout project. Tests have been done on the Model package as of now. The tests are done using one method for each class with a testing goal of 100% code coverage for each class methods.Test folder is currently placed at: /OOPP-HT20/app/src/test/java

5.1 Known Issues

1. **Issue 1**: Connecting back-end to the front-end through Android Native.
   - **Fix:** Everyone needs to become more familiar with Android Native.
2. **Issue 2**: Issues with creating Tests. Data can't be imported using Android specific methods. Currently using a workaround by creating test classes not using Andoid specific methods to get the Network data. This will result in incorrect data of tests ran in test folder. The missing methods is covered in testClasses for now.
   - **Fix:** Moving AssetsManager out of the Network class into a Services package.
3. **Issue 3:** Issues with adding an Image to a Report using Android Image. Just like the tests, we can't use Android specific methods to generate test data.
   - **Fix:** TBA.
4. **Issue 4:** In Network class, getPrevNode() and getNextNode() does not work as intended. Returns NullPointer?
   - **Fix:** TBA.

5.2 Test Results

Data on test results will be provided in folder OOPP-HT20/app/src/test/testresult.

5.3 Access control and security

No access control is implemented at the moment.

6 References