System Design Document for PlankaGBG

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

2020-10-22
version 1.1

1 Introduction

This document contains technical information regarding an application that documents the position of ticket controllers in a public transport network. The application is in this moment built as an Android application. The application searches for and displays information regarding the activity of ticket controllers, which has been previously reported by users of the application.

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 *GUI*.
4. **Controller**: 1: (**Code**) The package of the program that connects the *View* package to the *Model* package. 2: (**Model**) An individual responsible for checking whether or not a public transit passenger has a valid ticket.
5. **Network**: A central part of the *model* which contains and handles the various *stations* and *routes* in the system.
6. **Node**: *Stations* and *routes* are comprised of a number of *nodes*, each of which represents one stop (e.g ”Brunnsparken läge A”).

7. **Station**: An object in the *model* representing a particular location (e.g ”Korsvägen”, ”brunnsparken”).
8. **Route**: An object in the *model* representing the stops in a one-way trip from one *station* to another.
9. **Report**: When the user sees controllers the application lets the user compile the information and submit it as a *report*.
10. **Incident**: A compilation of multiple reports that take place on the same *route* or *station*.
11. **Reporter**: The user.
12. **RunTime**: The period where the program is running.
13. **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 a server application and a database relaying information to the client-side.

2.2 Classes

2.2.1 AbstractReport

An abstract class which serves as a base for Report objects. Contains the number of controllers that the user reports, a timestamp and possibly an image.

2.2.2 Network

A graph representing the public transport network in our model. Contains lists for both stations and routes, and is also responsible for creating, maintaining, traversing and updating the graph structure. The Network 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 Node

A representation of a singular stop in the traffic network. Contains the name of the stop along with a boolean used to track whether or not a report has been made regarding the stop.

2.2.4 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.

2.2.5 Station

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

2.2.6 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.

2.2.7 ReportRoute

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

2.2.8 ReportStation

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

2.2.9 Model

The model class acts as the primary connection between the model and the GUI.

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. The data format is as follows:

   Marklandsgatan A Bokekullsgatan A Högsbogatan A

4. Network then create Routes that should be in the network and adds them. The routes is created using the filename as key and the values read in order in the format shown above.

5. Network create Stations and adds every possible state a Station has. The station name is created using above data for each station minus the ”A” part. Then the different states is added to the list in the Station object.

6. Network will hold and manipulate any and all data.

2.4.2 RunTime

At runtime The Network is manipulated thru the MODEL class. The views communicate thru the following chain of classes.

$$Views- > mainactivity(controller)- > MODEL$$

There is no direct communication with there rest of the model at runtime all requests are forwarded thru the ”chain” above.

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:

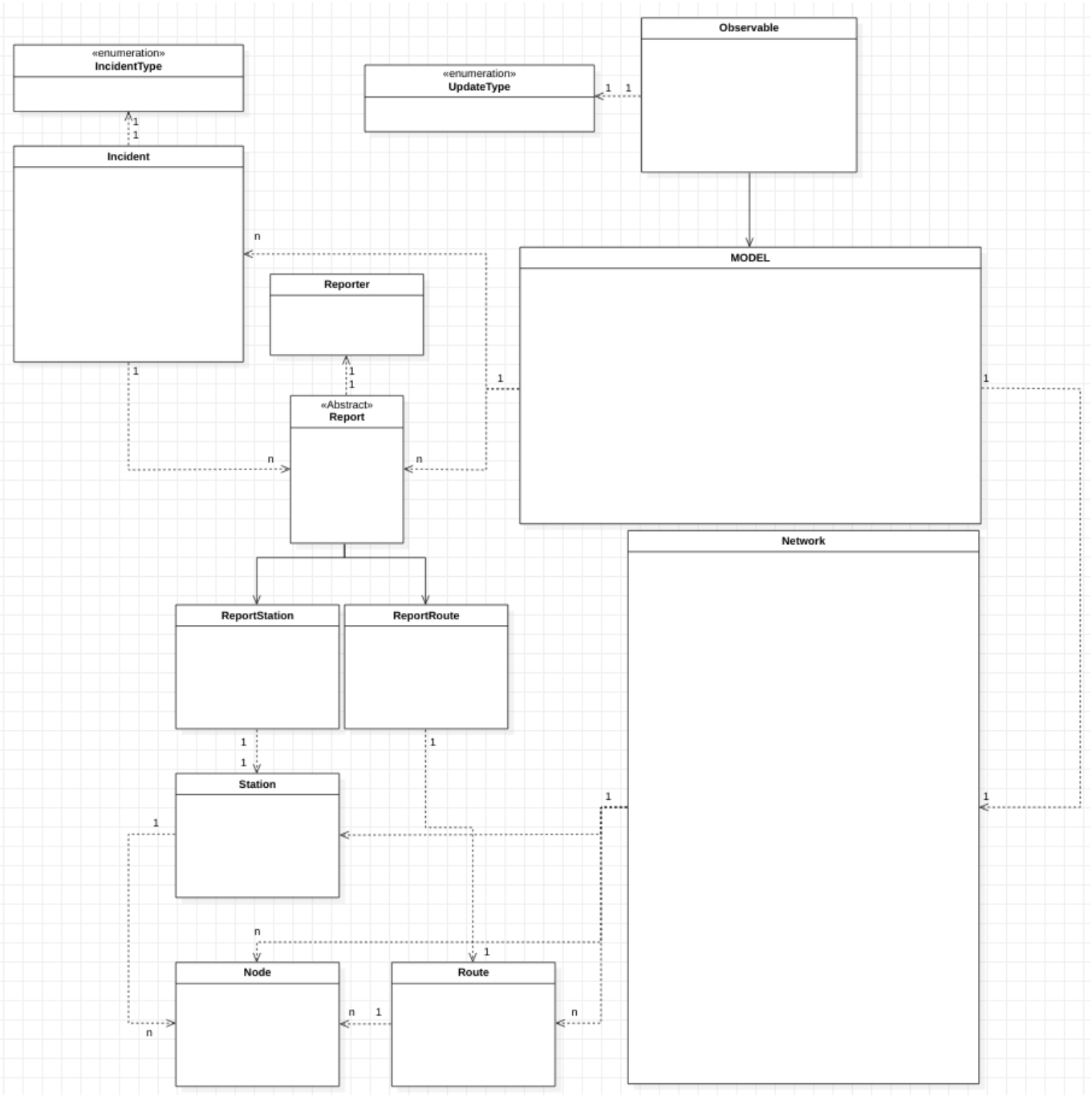

Figure 1: Domain Model

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).

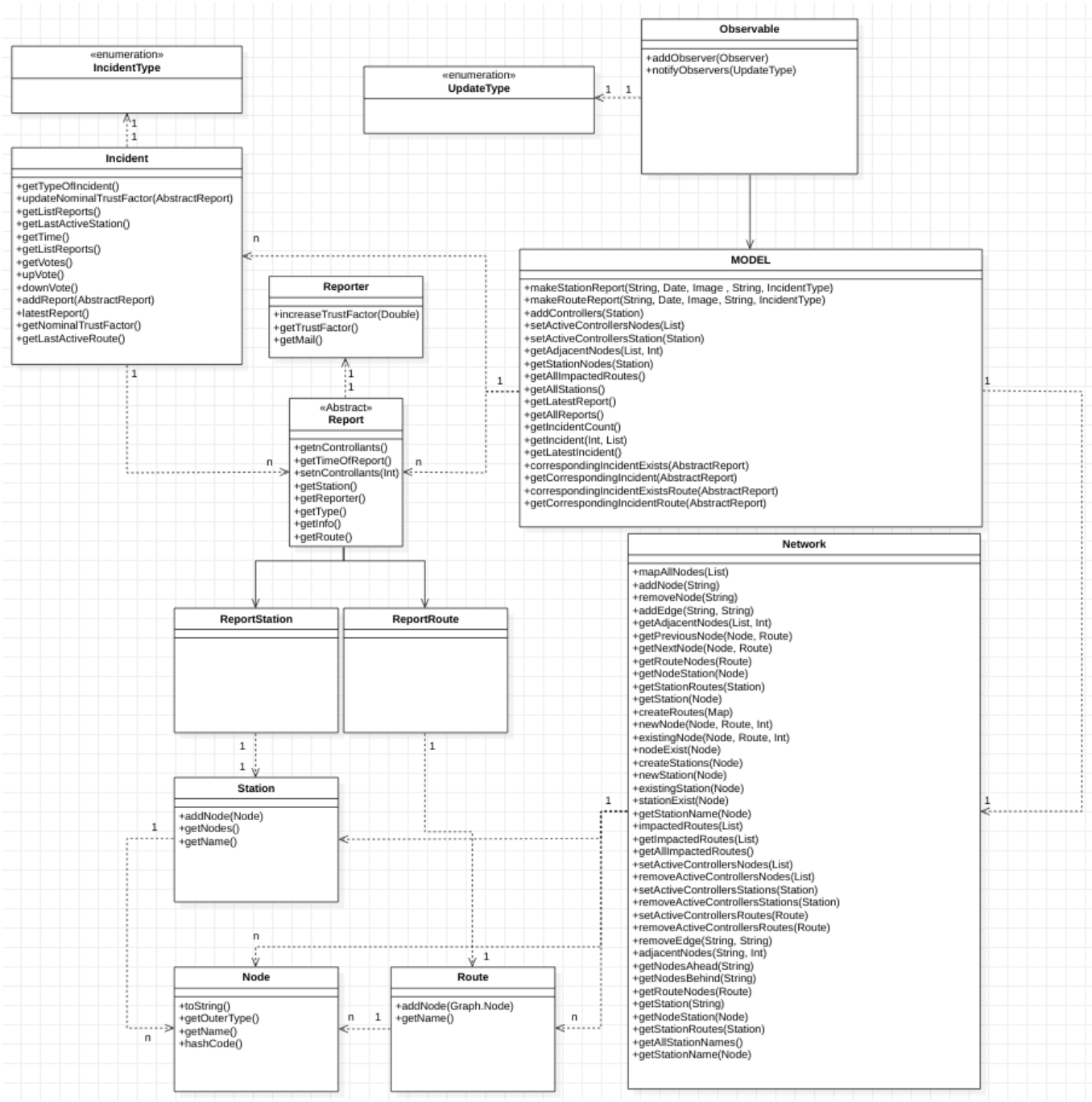

Figure 2: Design Model

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”. The .txt file consists of the name of a station and the stop on that specific station. The network class reads these files and creates routes and stations. The .txt files looks as followed:

```
Marklandsgatan A
Bokekullsgatan A
Högsbogatan A
Klintens Väg A
Godhemsgatan A
Mariaplan A
Ostindiegatan A
```

Figure 3: Example of a .txt file used by Network

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:** Issues with adding an Image to a Report using Android Image.
   - **Fix:** TBA.
2. **Issue 2:** In AbstractReport class, Type in abstract report makes type implementations generate null values instead of correct type.
   - **Fix:** TBA.
3. **Issue 3:** In Incident class, Method updateNominalTrustFactor doesn‘t adjust the value of trustfactor because of void-return in get-part for totalTrustFactor.
   - **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