# System Design Document for PlankaGBG

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

2020-10-23

## 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 the program that contains the *GUI*.
4. **Controller**:
    a) (**Code**) The package of the program that connects the *View* package to the *Model* package.
    b) (**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. Extends AbstractReport and contains a Route.

### 2.2.8 ReportStation

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

### 2.2.9 MODEL

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

### 2.2.10 FileReader

The FileReader class is responsible for reading input data from external .txt-files, which is used for creating Stations and Routes in the Network class.

## 2.3 Communication protocols

NA.

## 2.4 Description of program flow

### 2.4.1 Startup

1. Objects are initialized.
2. GUI is created.

3. FileReader reads in files containing information about every tram line. A more detailed description of the data files can be found under *Section 4: Persistent data management.*

4. Network creates all Routes that have been read by FileReader.

5. Network creates Stations and populates them by placing all connected stops (”Marklandsgatan A”, ”Marklandsgatan B” and so on) in a list in each object.

6. Network maps out all stops, and connections between stops, to create the entire tram network.

### 2.4.2 RunTime

At runtime the Network is accessed using the MODEL class. The various parts of the View communicate through the following chain of classes:

```
View(fxml) -> Mainactivity(controller) -> MODEL
```

There is no direct communication between the View and other parts of the model at runtime; all requests are processed using 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 and all relevant data exists in the application.

## 3 System design

We have decided to use Model-View-Controller.

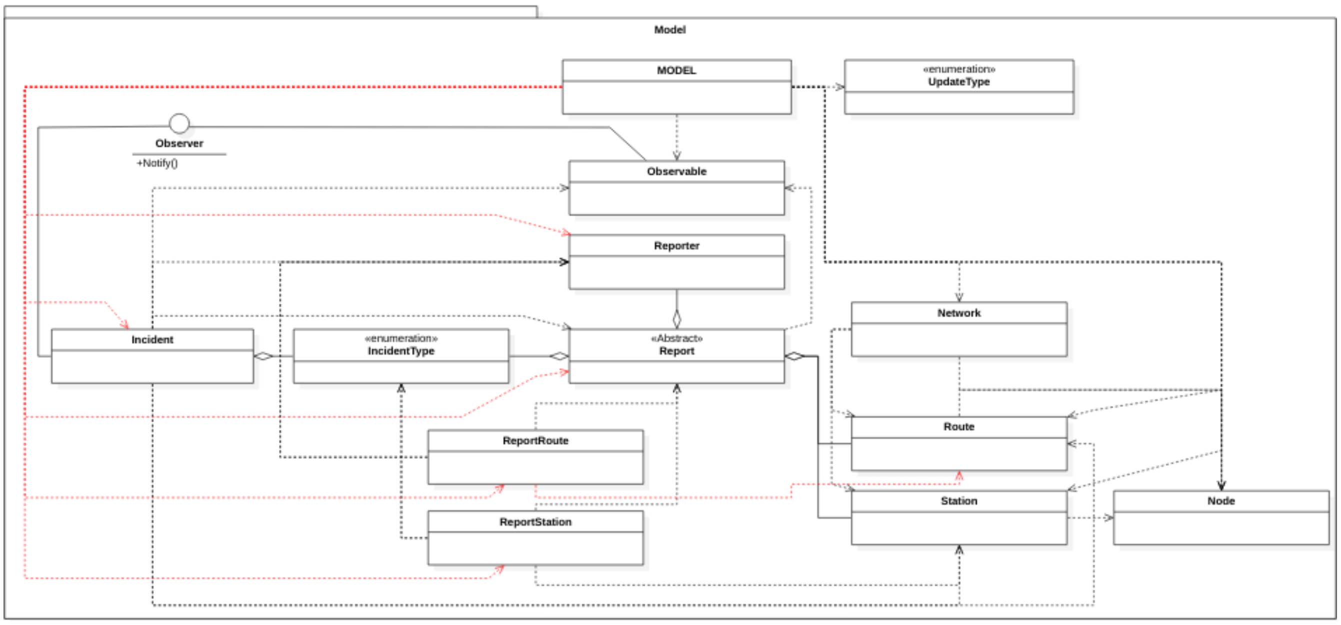


Figure 1: Model Package

This is the package that is responsible for all the logic in the application. It is responsible for mapping out the tram network, traversing the tram network, creating Route and Station Objects as well as handling all User Information and the creation of Reports.

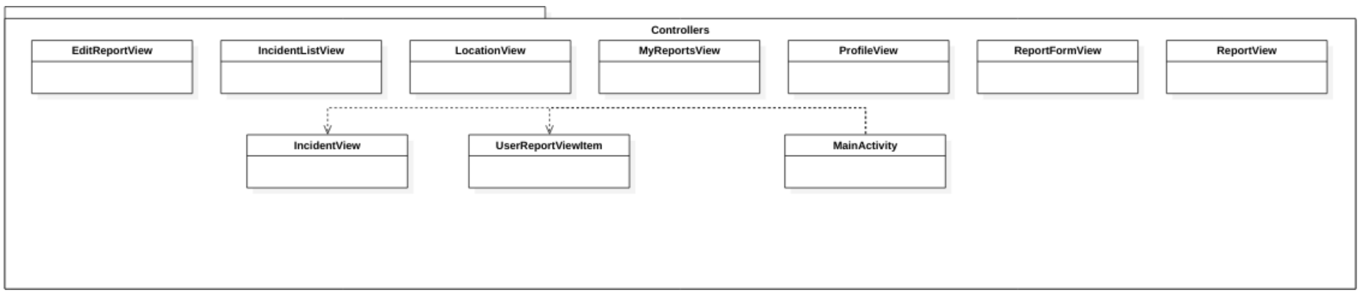


Figure 2: Controllers Package

This is the package that is responsible for interfacing the backend with the GUI.

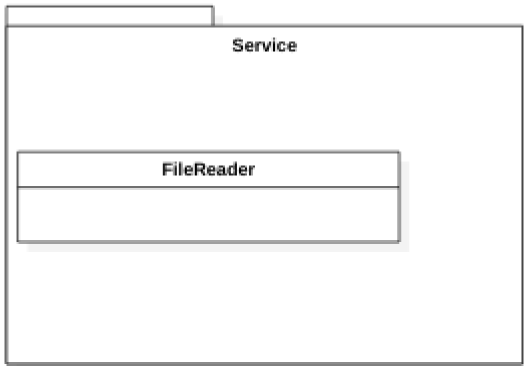


Figure 3: Service Package

This package contains all Services used by the Application. In this case, a FileReader, that is used to read files stored under ”Assets” in the application. These output from these files are then used to create the tram network.

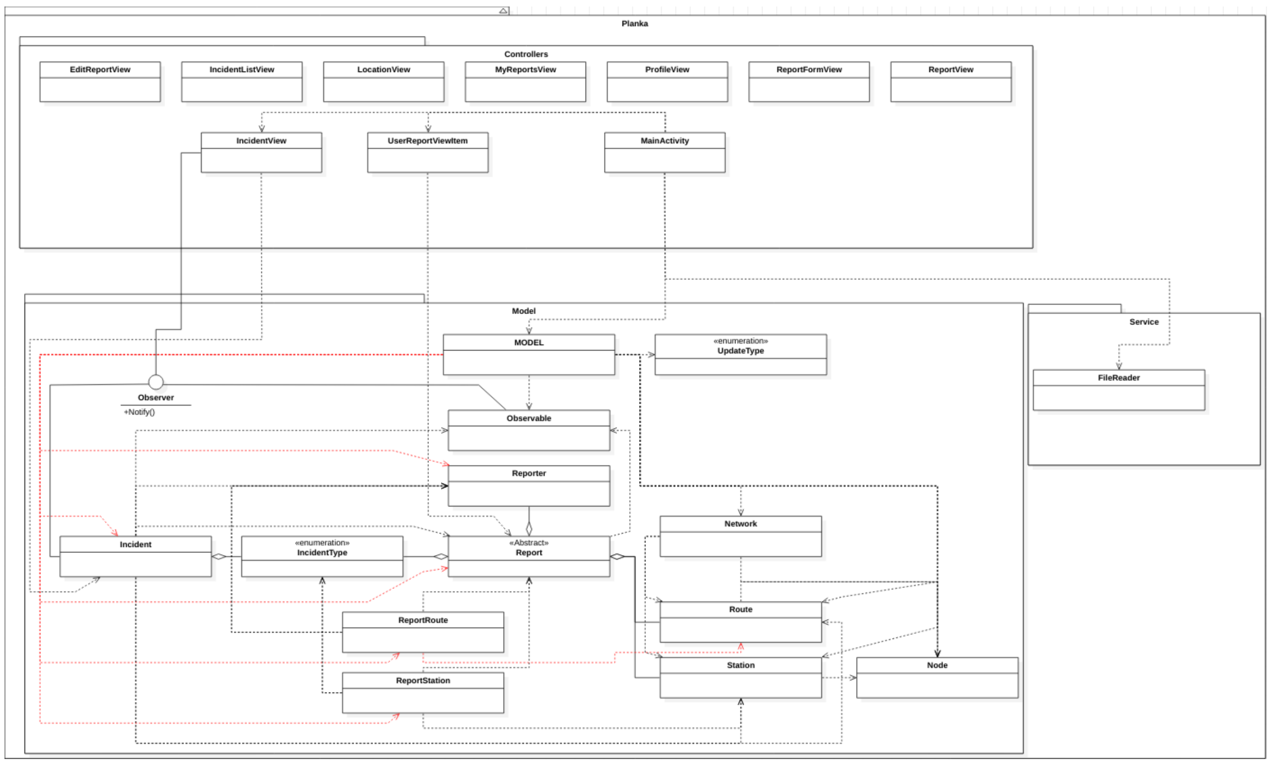
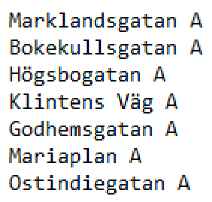


Figure 4: Package Coupling

This is all packages, how they are interconnected and interact with each other, in the application.

## 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 a list of stops along a given route. The FileReader Class reads these files, which are then sent to the Network class to create the Route and Station Objects as well as mapping out the tram network. The .txt files are formatted as follows:



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

Figure 5: Example of a .txt file used by Network

## 5 Quality

Quality assurance consists of tests and code reviews. Testing is done using JUnit-test and these tests will be provided in a test-directory. The code is continuously tested throughout development but focus have been on the Module package. The tests are done using test classes that are copies of the classes in the Model package. Abstract classes are tested indirectly by testing the functionality in the subclasses, with a testing goal of 100% code coverage for the methods in each class.

### 5.1 Known Issues

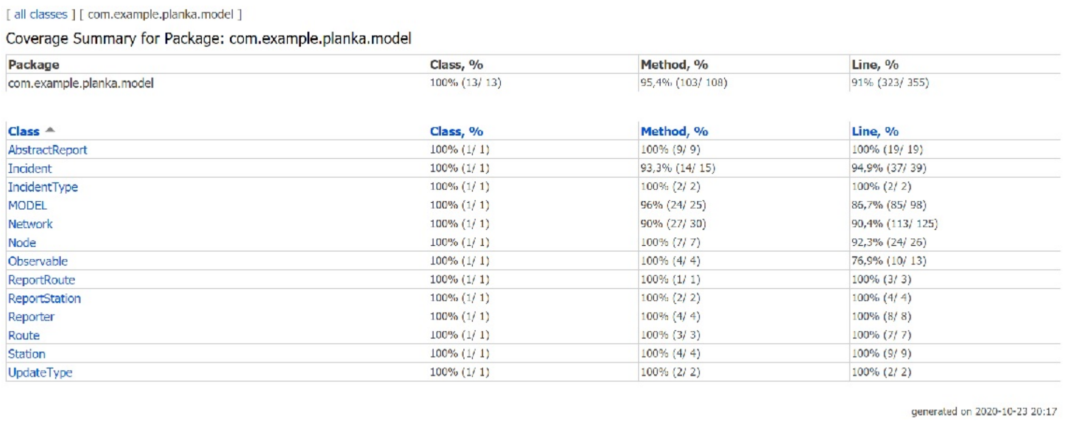
NA.

### 5.2 Test Results

The tests will be found in the following directory:

```
/OOPP-HT20/app/src/test/java
```

The test results looks like the following:



[ all classes ] [ com.example.planka.model ]

Coverage Summary for Package: com.example.planka.model

| Package | Class, % | Method, % | Line, % |
|---|---|---|---|
| com.example.planka.model | 100% (13/ 13) | 95,4% (103/ 108) | 91% (323/ 355) |

| Class ▲ | Class, % | Method, % | Line, % |
|---|---|---|---|
| AbstractReport | 100% (1/ 1) | 100% (9/ 9) | 100% (19/ 19) |
| Incident | 100% (1/ 1) | 93,3% (14/ 15) | 94,9% (37/ 39) |
| IncidentType | 100% (1/ 1) | 100% (2/ 2) | 100% (2/ 2) |
| MODEL | 100% (1/ 1) | 96% (24/ 25) | 86,7% (85/ 98) |
| Network | 100% (1/ 1) | 90% (27/ 30) | 90,4% (113/ 125) |
| Node | 100% (1/ 1) | 100% (7/ 7) | 92,3% (24/ 26) |
| Observable | 100% (1/ 1) | 100% (4/ 4) | 76,9% (10/ 13) |
| ReportRoute | 100% (1/ 1) | 100% (1/ 1) | 100% (3/ 3) |
| ReportStation | 100% (1/ 1) | 100% (2/ 2) | 100% (4/ 4) |
| Reporter | 100% (1/ 1) | 100% (4/ 4) | 100% (8/ 8) |
| Route | 100% (1/ 1) | 100% (3/ 3) | 100% (7/ 7) |
| Station | 100% (1/ 1) | 100% (4/ 4) | 100% (9/ 9) |
| UpdateType | 100% (1/ 1) | 100% (2/ 2) | 100% (2/ 2) |

generated on 2020-10-23 20:17

Figure 6: Current test coverage

Structure analysis done using Stan4J on our application looks like this:

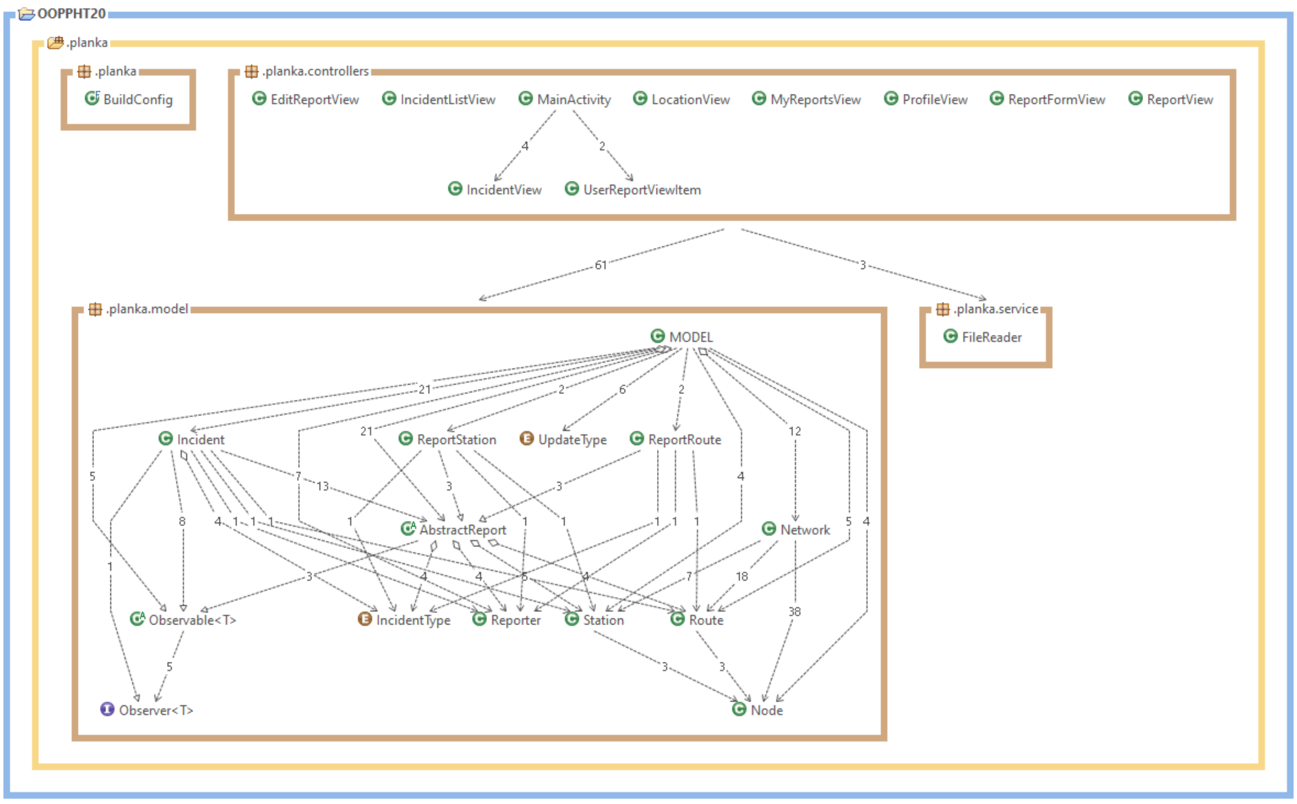


Figure 7: Structure and dependency analysis created in Stan4J

### 5.3 Access control and security

NA.

## 6 References