Requirements and Analysis Document for PlankaGBG

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

2020-10-23

1 Introduction

The purpose of the application is to report the current positions of ticket controllers in the Gothenburg public transport network and display this information to users in near real-time. The reports are created by users of the application, after which other users may choose to vouch for the credibility of the report if they deem it to be correct. Users may also choose to receive a notification when controllers are reported along a route of their choice.

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 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. **Incident**: Multiple reports that takes place on the same route or station.
6. **Report**: When the user sees controllers the application lets the user compile the information and submit it. A report.

7. **Reporter**: The user.
8. **Non-functional requirement**: Requirement that defines how a system should be.
9. **Functional requirement**: Requirement that defines what a system should do.
10. **Java-doc**: Java-doc is a documentation generator that generates HTML documents from java source code.

2 Requirements

2.1 User Stories

Story Identifier *US1* - **Make a Report - Implemented**

1. Description:
   - As a User, I want to be able to report when I see controllers so that other users can see where the controllers are.
2. Acceptance Criteria:
   - Users can create Reports.
   - Reports are always connected to an Incident when created.
     - If no matching Incident exists, a new one is created alongside the Report.
   - The Incident is updated and displayed in the GUI.
3. Functional Requirements:
   - Can I click a button to make a Report?
   - Can I make different kinds of Reports?
4. Non-functional Requirements:
   - **Reliability:** Application should not crash when creating a new Report.
   - **Usability:** It should be easy for the User to create a new Report.

Story Identifier *US2* - **Endorse Incident - Implemented**

1. Description:
   - As a User, I want to be able to endorse an Incident so that other Users can see that it is trustworthy.
2. Acceptance Criteria:

- An Incident can be endorsed by users.
- Said Incident is deemed more trustworthy if a User reports it as such.
- The incident is updated and displayed in the GUI.

3. Functional Requirements:
   - Can I confirm that an Incident is correct?
4. Non-functional Requirements:
   - **Reliability:** Application should not crash when endorsing an incident.
   - **Usability:** It should be easy for the User endorse an incident.

Story Identifier *US3* - **Search and Browse the Network - Implemented**

1. Description:
   - As a User, I want to be able to see where the controllers are so that I can be more careful in my travels.
2. Acceptance Criteria:
   - Recent incidents should be visible in the GUI.
   - A User should be able to search for specific stations and routes to see if they are affected by recent incidents.
3. Functional Requirements:
   - Can I click on a button to search for Incidents related to a tram line?
   - Can I see recent Incidents in a the GUI?
4. Non-functional Requirements:
   - **Scalability:** It should be easy for Developers to scale the functionality.
   - **Usability:** It should be easy for the User browse the controllers in the Network.

Story Identifier *US4* - **Identifying Image - Not Implemented**

1. Description:
   - As a User, I want to be able to attach an image to my report to further strengthen its credibility.
2. Acceptance Criteria:
   - If an image is uploaded, it is displayed alongside the incident it is attached to.
   - The credibility of the incident and report is strengthened if an image is attached.

3. Functional Requirements:
    - If I create an Report, can I add an image?
    - Can I change the image afterwards?
4. Non-functional Requirements:
    - **Usability:** It should be easy for the User attach an image to a report.

Story Identifier *US5* - **Trust Factor - Implemented**

1. Description:
    - As a User, I want to have a trust factor that impacts the credibility of my Reports.
2. Acceptance Criteria:
    - The trust factor of a user is increased when a Report they've made is backed up by other users or reports.
    - When a Report is created, an auto generated trust factor is assigned to that Report based on the trust factor of the user.
3. Functional Requirements:
    - Can the trust factor of a user be altered?
    - Can a user impact the trust factor of other users?
4. Non-functional Requirements:
    - **Security:** One User should not have direct access to the private information of another User.
    - **Reliability:** Trust factor should not be changed randomly for each user.

Story Identifier *US6* - **Modify Report - Implemented**

1. Description:
    - As a User, I want to be able to change or modify a Report after it has been made.
2. Acceptance Criteria:
    - After a Report has been made, the reporter should be able to modify it.
    - The Report is updated when a modification is made.
    - The updated Report is displayed in the GUI.
3. Functional Requirements:
    - Can I add or remove something from a Report I have created?

4. Non-functional Requirements:
    - **Security:** One User should not have direct access to another User's report, meaning they should not be able to change or modify it.
    - **Reliability:** A report should not be altered unless the User modifies it.

Story Identifier *US7* - **Notification - Implemented**

1. Description:
    - As a User, I want to be notified if a controller is on my route and/or near me.
2. Acceptance Criteria:
    - Users should have the option to receive notifications.
    - Users should be notified about controllers on their route, or near them, if they have said option enabled.
3. Functional Requirements:
    - Can I enable a notification function?
    - Is the notification function affected by Reports?
4. Non-functional Requirements:
    - **Manageability:** The User should easily be able to turn the Notification on and off.
    - **Reliability:** The application should not crash when sending the User notifications.

Story Identifier *US8* - **Map - Not Implemented**

1. Description:
    - As a User, I want a graphical representation of where reports has been made.
2. Acceptance Criteria:
    - Users should be able to access a map that displays currently active Reports.
    - The Map should correctly display where and when Reports were made.
3. Functional Requirements:
    - Can I see where Reports has been made on a map?
    - Can I see when the Reports were made?
4. Non-functional Requirements:
    - **Manageability:** The User should easily be able to switch to the Map view.
    - **Reliability:** The Map should not force the User to close the app to switch view.

2.2 Definition of Done

The entire code should be commented using Java-doc standard, methods and variables should be declared in such a way they are easy to understand and the code should have been tested and peer-reviewed by the group. Any visual elements that are part of the code should be implemented in the GUI of the application. After every feature branch has been approved they should be merged to master to allow for continuous development and integration.

2.3 User Interface

Screenshots from our GUI can be seen in Figures 1 through 5 below. The design of the application has been purposefully kept simple to make it easy to use under stress, as we reckon that people tend to be heavily stressed while using the public transport network.The GUI consists of three primary ”tabs”. Navigation is handled using a toolbar at the top of the screen.

2.3.1 Incidents

The Incident List View [Figure 1] displays all currently active incidents in Gothenburg. Each incident features information regarding its location, the time at which the incident occurred as well as the total number of reports. This view also allows the user to search for specific routes and stations, to see if any incidents currently affect them.

2.3.2 Reporting

The Report View [Figure 2] allows users to fill in and report information regarding incidents they witness, after which the Live Reports View is updated. This view also lists all reports that the user has made and allows recently made reports to be modified [Figure 3 and Figure 4, respectively].

2.3.3 Personal Information

The Profile View [Figure 5] allows the user to change which email address is associated with their account and lets the user opt into receiving a push-notification when an Incident is reported on a tram line of their choice.

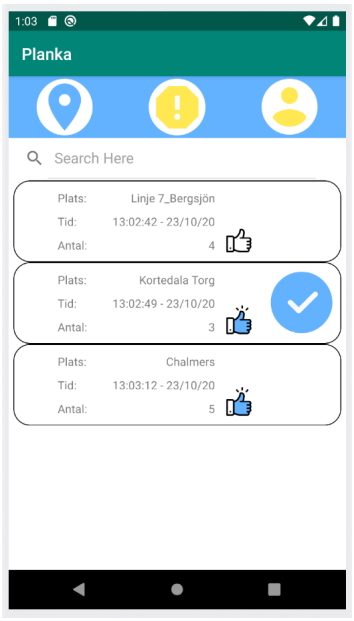

Figure 1: Incident List

Figure 2: Report Creation

Figure 3: User Reports

Figure 4: Report Editing

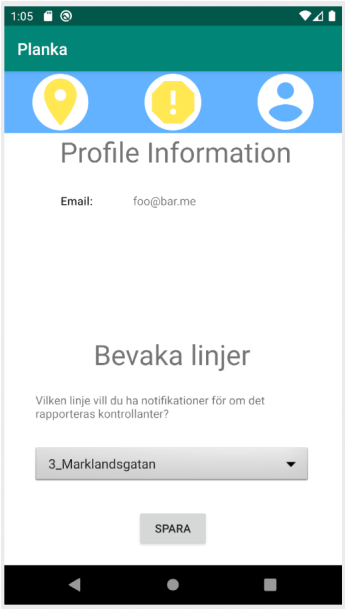

Figure 5: Profile

3 Domain model

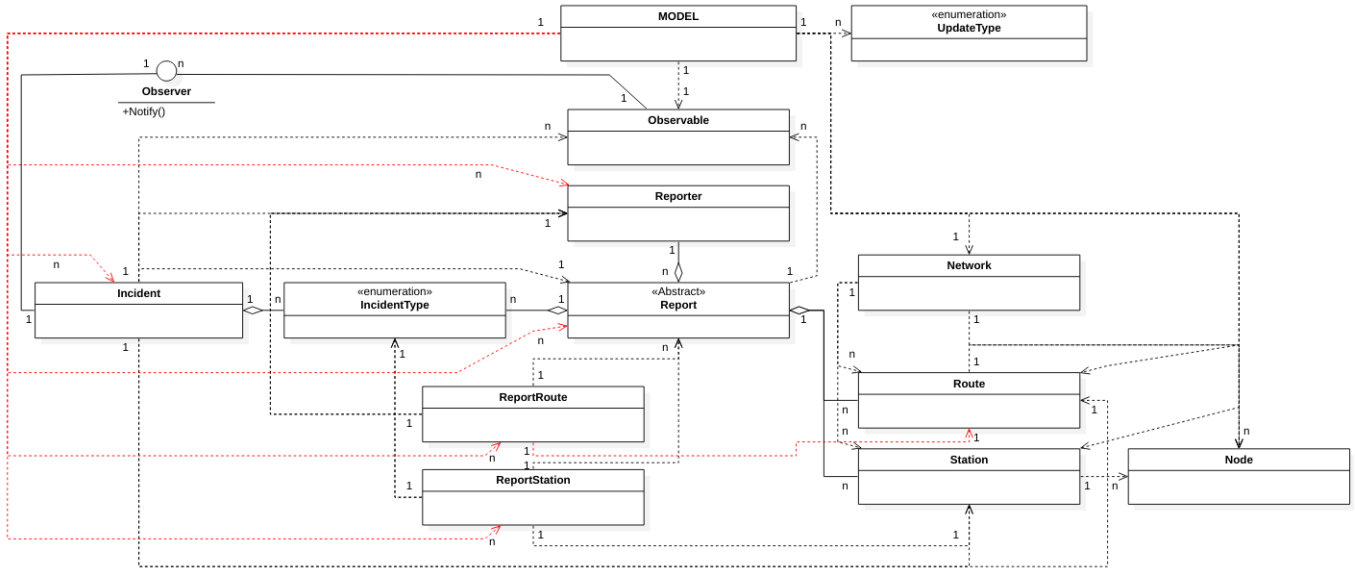

Figure 6: Domain Model

3.1 Class responsibilities

Explanation of responsibilities for the classes in the domain model above:

1. Node
   - Class responsible for each tram station in the network.
   - Contains information such as name and alarm state.
2. Route
   - Class responsible for each tram line in the network.
   - Takes a list of Nodes, one Node for each station the tram passes by.
3. Network
   - Class responsible for the entire tram network and its connections. Built up by adding every route to the data structure.
4. Station
   - Class responsible for representing the stations in our tram network, a station contains a list of individual stop positions for a station.

- For example, Station Centralstationen contains Centralstationen A, Centralstationen B etc.

5. Reporter
   - Class responsible for representing the User in a report.
   - Protects the Users integrity and only contains an email address and a factor for how trustworthy the User is.
6. AbstractReport
   - Abstracts content from the RouteReport class and StationReport class and represents the reports for the Incident class.
7. RouteReport
   - Class responsible for Reports that involve a Route.
8. StationReport
   - Class responsible for Reports that involve a Station.
9. Incident
   - Class responsible for gathering Reports that all involve the same Station or Route.

4 References

List all references to external tools, platforms, libraries, papers, etc.

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.

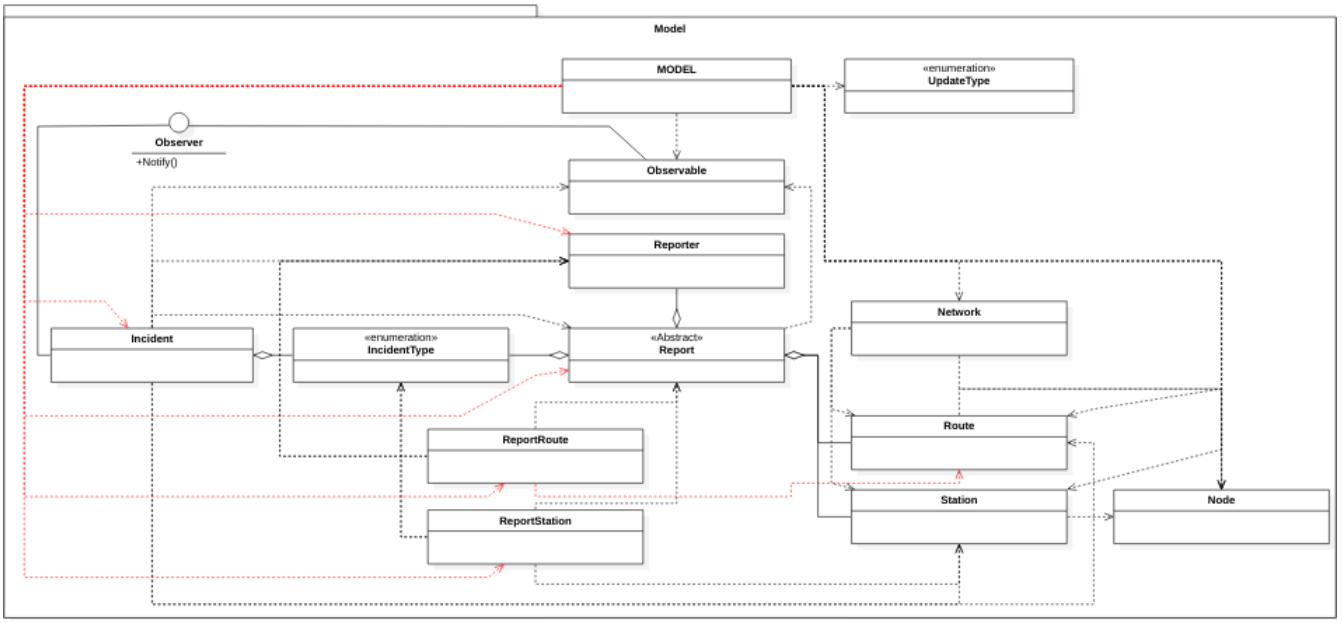

Figure 1: Model Package

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.

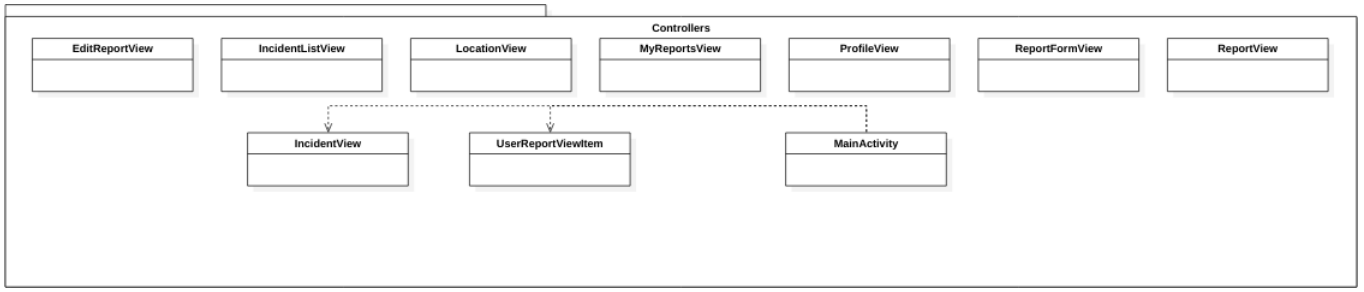

Figure 2: Controllers Package

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

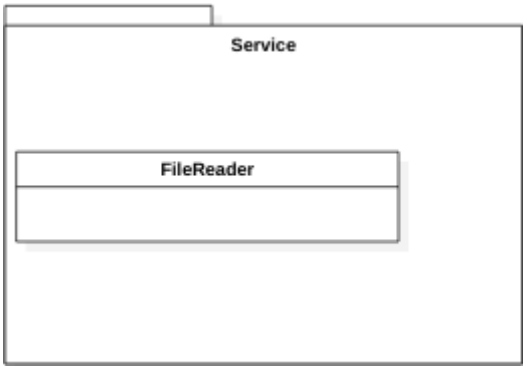

Figure 3: Service Package

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.

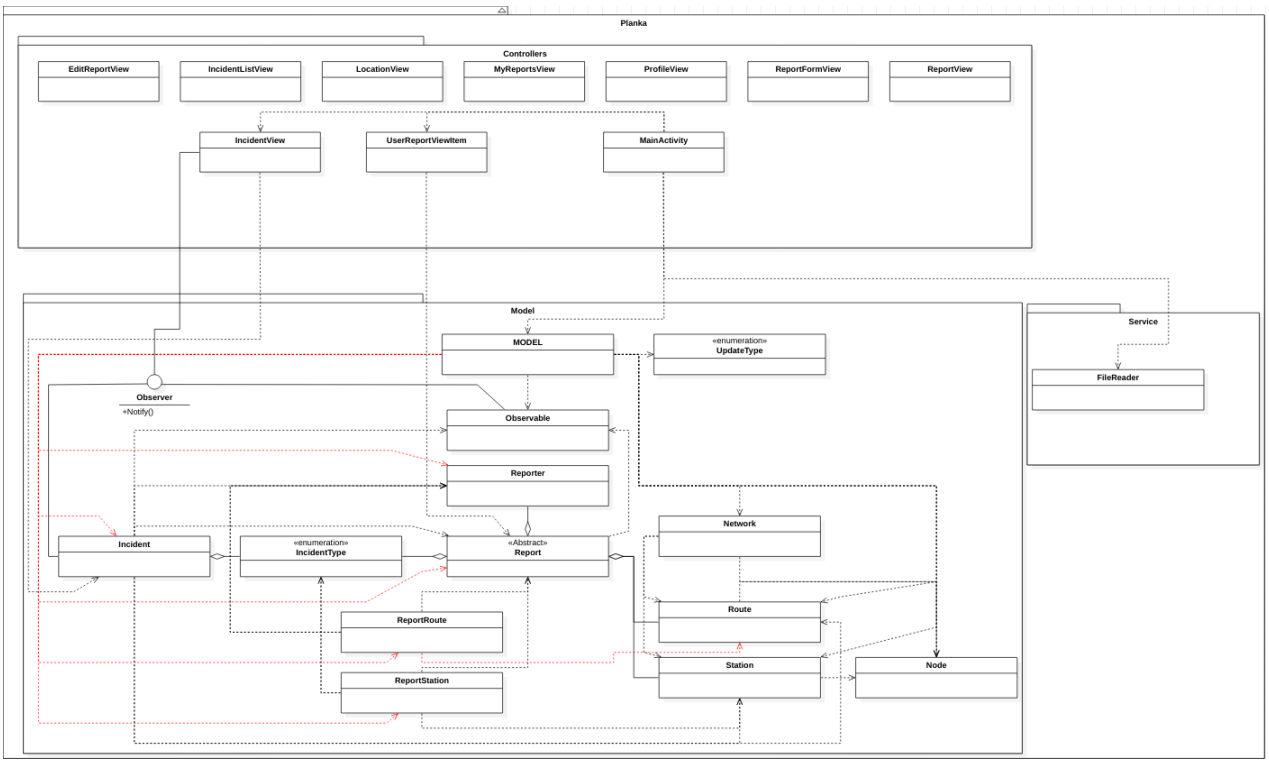

Figure 4: Package Coupling

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:

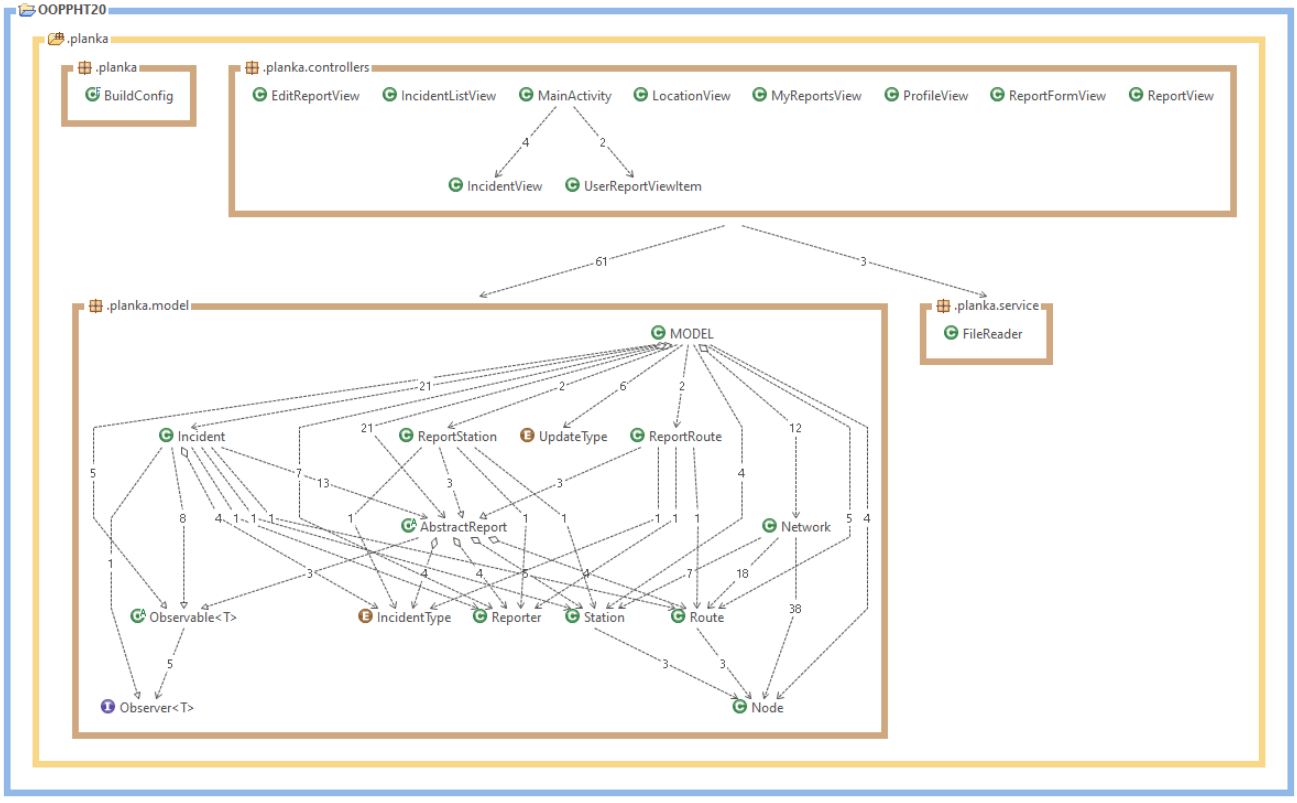

Figure 7: Structure and dependency analysis created in Stan4J

5.3 Access control and security

NA.

6 References

Peer review för ObPaint

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

2020-10-09

1 Peer review

1.1 Designprinciper och designmönster

Model View Controller utgör i det stora hela nästan all kod. Alla klasser, exklusive tester, är placerade så att de uppfyller mvc pattern. Även beroenden är bra, koden undviker onödiga beroenden mellan model, view och controller.

Ni använder även factories på flera ställen i koden, specifikt för att skapa ConcreteShape, ConcreteTool samt respektive interface. Detta är ett korrekt sätt att implementera factory pattern i er kod och finns inte något att anmärka på.

Modellen i ObPaint använder singleton designmönstret flitigt och är skriven som en enum istället för en ”vanlig” klass. Detta är bättre för ändamålet då enum implicit är begränsat till en instans samt inte har några problem med att onödiga instanser skapas vid serializing och de-serializing, snyggt!

Ett Observer-pattern används för att projektets modell ska kunna notifiera CanvasController när det valda ritverktyget ändras. Detta har implementerats väl och verkar vara en resonlig lösning till problemet.

1.2 Dokumentation

Dokumentationen är konsekvent, tydlig och täcker i stort sett all relevant kod förutom testklasserna. Lite mer dokumentation för varje metod skulle kunna vara nödvändigt, då det hade gjort det enklare för personer utanför projektet att snabbare sätta sig in koden.

1.3 Kod

Det finns flera metoder som returnerar ett värde av null, vilket gör att viss funktionalitet inte finns. Detta är däremot redan noterat i dokumentationen. I övrigt så har koden skrivits i en konsekvent stil. Alla namn på variabler etc. är också det väldigt bra, tydliga och förklarande om vad variabeln, metoden och klassen innehöll för data. Både vid överblick och noga genomgång av koden så ser allt faktist väldigt bra ut, ni har skött beroendena och konstruktionen av klasser snyggt. Däremot finns det flertal klasser som de har valt att kommentera ut innan de lämnade in projektet för peer-review. De ger oss känslan att klassena kanske inte bör vara där från första början.

1.4 Testning

Testningen som finns är bra men skulle kunna vara betydligt mer omfattande då de saknas en hel del. En bra tanke är att utgå ifrån att alla metoder ska testas(coverage), möjligen att getmetoder kan undantas. Saknas även kommentarer om vad som testas i testmetoderna.

1.5 RAD

Bra med prioritet av user stories. User story f18 är inte klar. Saknar förhållanden mellan klasserna, t.ex. att 1 ModelCanvas kan innehålla *n* antal Shapes. Hade också varit bra om det, där det är relevant, fanns beskrivande ord i pilarna, t.ex. ModelCanvas *shows* Shapes. Favorites nämns i domänmodellen men återfinns inte i koden eller SDD:ns systemdesign?

1.6 SDD

Projektets SDD ser bra ut och har en tydlig struktur och uppdelning. Bra med en tydlig beskrivning om vad programmet ska åstadkomma. Något som saknas är dock en beskrivning av vad som har gjorts angående säkerhetsaspekten t.ex om det ska implementeras någon form av användarkonton, om dessa kommer ha någon form av hierarki m.m.

1.7 Runtime

Programmet kan köras utan problem och ser bra ut med en tydlig och genomtänkt GUI. Det enda vi kunde upptäcka som går att anmärka på här är att man kan skapa former som hamnar utanför canvasen och det ser lite roligt ut, men det är antagligen något ni redan vet om.