Developer Documentation for Waft

1 Introduction

Waft is an Android application with the purpose to enable commuters to share information with each other regarding public transport. This goal is to give commuters more information about what's happening on buses, trams and trains, and also to be able to gather data about how the commuters experience public transport.

The application consists of two parts - a server written in Node.js, and an application written in Java that requires an Android SDK version of 21 or higher. The server stores data that is generated in the application, and enables other instances of the application to use it.

This document describes the system design of the application. For documentation regarding the server, see https://github.com/Oscmage/DAT255_server/tree/master/docs. The design goals described in this document also apply to the server.

1.1 Design goals

The main goal of the software is to be able to replace parts without doing major changes to the code. The code should be separated into distinct packages and classes, where each package and class should have a clear purpose. The design should also allow for easy testing of separate components.

The internal documentation of the code should properly describe the purpose of both classes and methods, and also how they are used.

The code should be clean, and using libraries and built in functionality where it can be used is a priority. It is more important to have a cleaner codebase than to support a larger amount of phones.

1.2 Definitions, acronyms and abbreviations

- **GUI**: Graphical User Interface. The view that is shown to the user.
- **Android SDK**: Android Software Development Kit. A set of tools for developing Android applications. The Android SDK has version numbers, where a higher number indicates a later version of the SDK. Older phones don't support the newer versions of the SDK.
- **Node.js**: A JavaScript runtime for servers. This enables the use of JavaScript on a server.
- **Vehicle**: A vehicle in the public transport system, for example a bus, tram or train.
- **Flag**: A user generated piece of information associated with a vehicle.
- **MVC**: Model View Controller. A way to structure code that effectively separates the view code from the domain logic.
- **DIP**: Dependency Inversion Principle, a software pattern. Dictates that high level modules should only depend on abstractions of low level modules, and details should depend on abstractions.

2 System architecture

2.1 Overview

The application code is structured according to the Layered Architecture pattern (Evans (2004, p. 68-75)), which is a modified form of MVC. This orders packages hierarchically, meaning that higher level packages may have knowledge of lower levels, but not vice versa. Lower levels communicate with the higher ones through the observer pattern (Gamma, Helm, Johnson and Vlissides, 1995).

The following hierarchy is used, where lower number means a higher package level.

1. Presentation
2. Application
3. Domain (or Model)
4. Infrastructure

The presentation layer roughly has functionality that corresponds to the View and Controller parts of MVC. This layer contains everything that has to do with the user interface, both the looks of it and how it behaves.

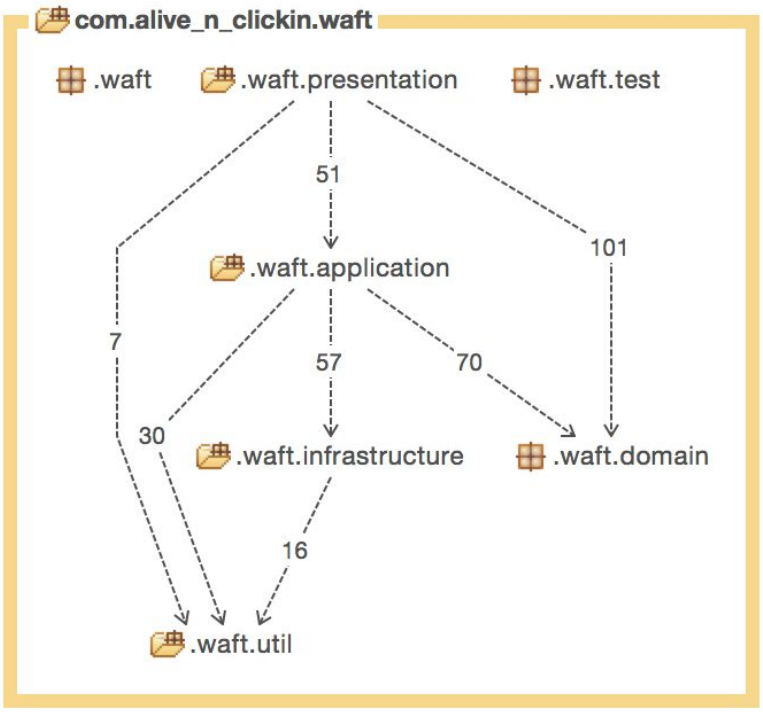
The application layer handles application state, and connects the presentation layer to the infrastructure layer. The presentation layer goes to the application layer to retrieve the data it needs for its views.

The domain layer contains all business logic of the application. This layer only contains plain Java objects and has no internal dependencies. Note that the domain layer is allowed to have knowledge of the classes in the infrastructure layer, but at present, there is no need for such knowledge, so the infrastructure layer and domain layer are completely independent at the time of writing.

The infrastructure layer is for handling low level tasks such as connecting to databases and API:s. The infrastructure layer allows the other layers to interact with lower level details through abstractions, keeping the code higher up in the hierarchy pristine and focused.

There is also a `util` package. This package is for utility classes that can be used by all other packages. It holds useful tools that are not directly connected to the project, but acts more as our own help library of classes that are handy to have.

Most classes have interfaces, and these interfaces are used to describe the type of an object throughout the whole application. This is a way of using DIP to ensure that classes are loosely coupled, and to make it easy to replace specific implementations.



3 Design solutions

Apart from the architecture described above, the application uses a few design choices that should be obeyed and understood to keep the code quality high in the future as well. Familiarize yourself with these, and your understanding of the system will increase manyfold.

3.1 Domain model

All classes in the domain model are immutable. This makes it easy to handle them in the other layers, since the objects can't be modified after instantiation. The objects can be shared freely without fear of unwanted state changes. They can be created and discarded as needed. They are only representations of data, and have no identity beyond this data.

The domain model in Waft is rather thin, since the business logic is simple and straightforward. The domain layer contains classes representing bus stops, vehicles, and flags.

3.2 User interface

The user interface is split up into two separate types of files, XML files and Java files.

3.2.1 XML

The XML files are a static structuring of the interface. There are layout files that define the style and ordering of UI elements. There are also files that holds string and integer values, references to images, style themes, and more. This is a standard way of structuring an Android application. For a detailed description of how Android uses XML, we refer you to the many tutorials and resources that are readily available online.

3.2.2 Java

The Java classes of the user interface (found in the presentation package) act as a kind of controllers for the views. They modify the user interface, add listeners to clickable items, and so forth.

Still, these classes only pertain to the user interface. They don't directly modify the domain (though they read data from it) or perform business logic. Rather, they forward these kinds of tasks to the application layer.

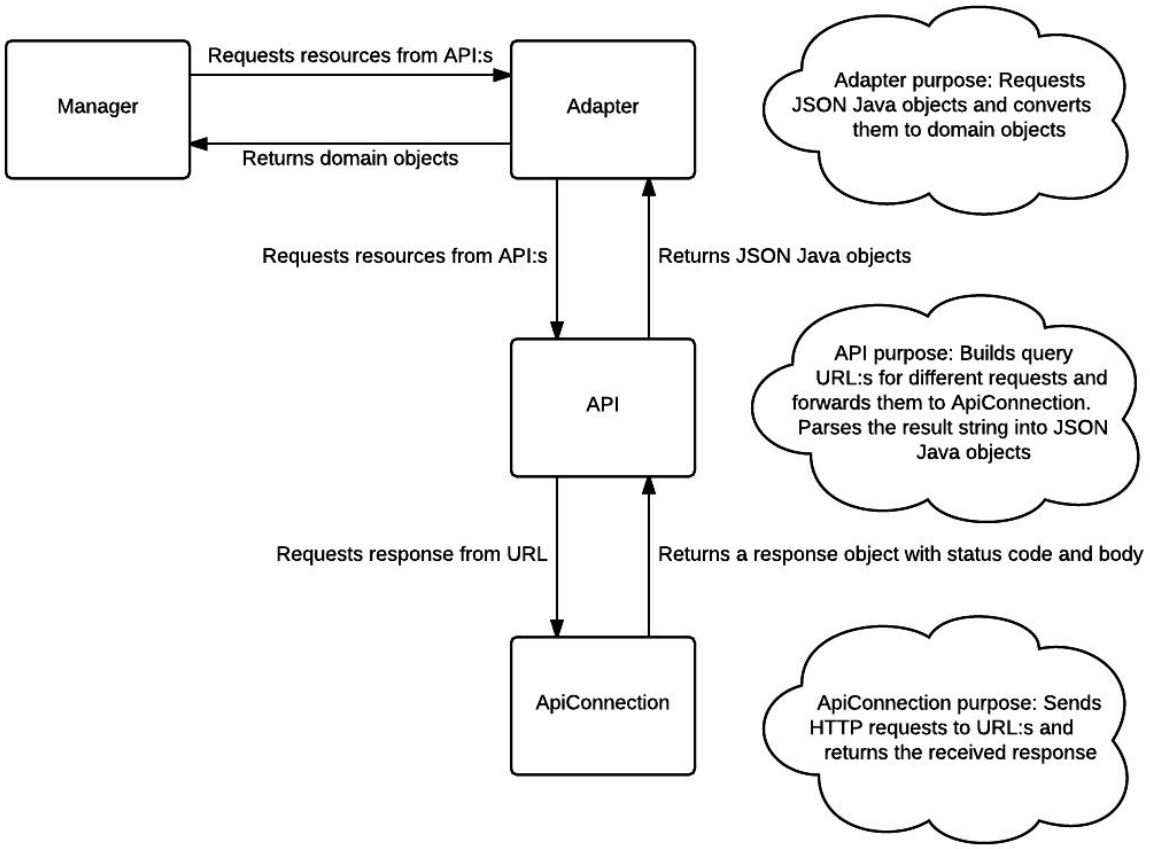
The packages in the presentation layer are ordered into submodules according to which activity they relate to. There are two activities in the app. One for searching for vehicles heading to a certain stop, and one for adding flags to the vehicle you're on.

3.3 Manager

The manager class is the main gateway for the presentation layer to the rest of the application. This class has high level methods that the presentation classes can use to perform tasks such as figuring out which bus the user is on, flagging buses, finding stops and so forth. The manager class forwards the details of these tasks to other parts of the application that are specialized to handle it. That way, the presentation layer doesn't have to know very much about the rest of the application.

It might help to think of the manager as a "controller for controllers". A delegator that has overview over the application packages and "manages" it, the way an office manager might manage their team.

3.4 Talking to API:s



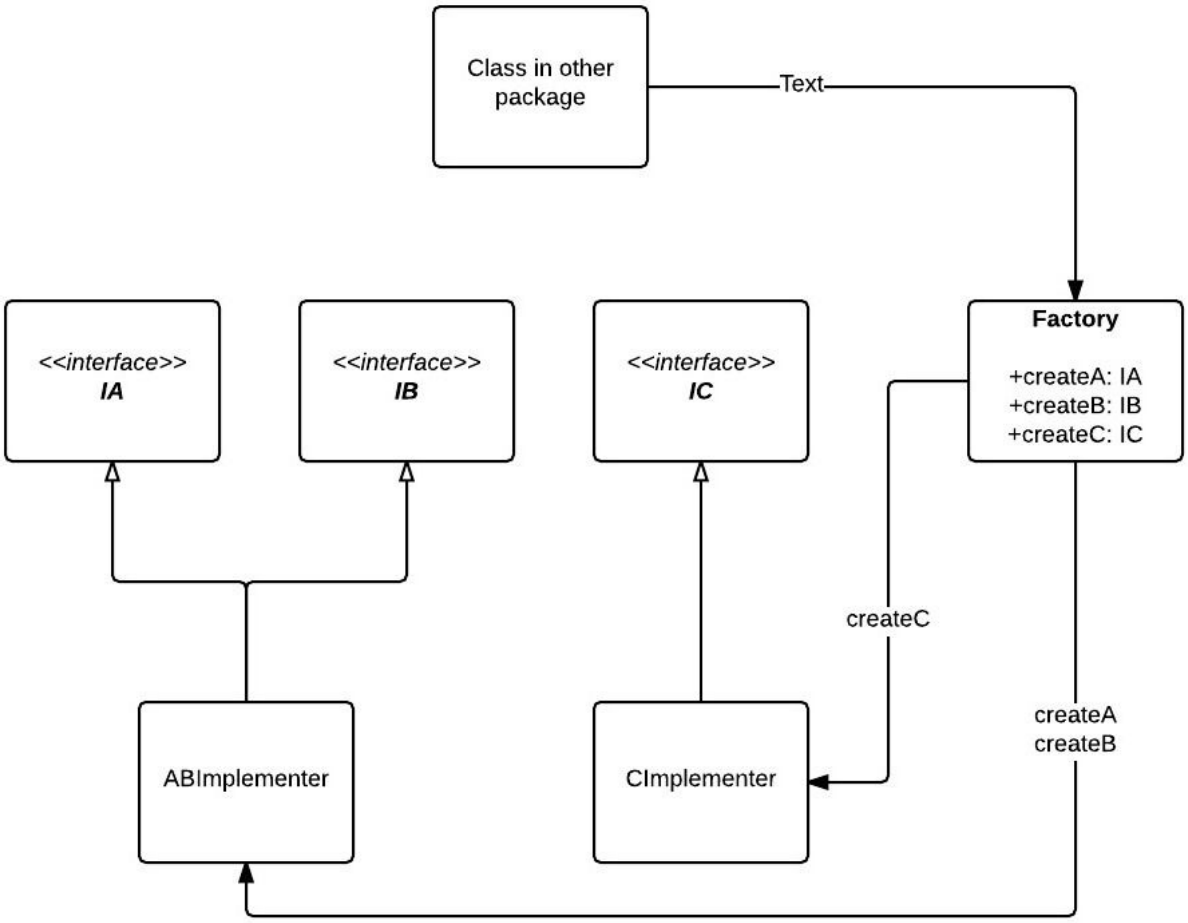
When communicating with API:s, a number of different classes are used. ApiConnection has static methods for sending GET, POST, and DELETE requests to specified URL:s with specified parameters. There are a few classes that are suffixed with “Api” (VasttrafikApi, ElectriCityApi, WaftApi) that builds correct query strings for requests, forwards them to ApiConnection, and parses the received results into Java objects that represent the JSON data format that was received. The adapter classes are responsible for converting these JSON data objects to domain objects, so that the API response can be used by the rest of the application.

This structure splits up the complicated process of talking to API:s that includes building queries, sending HTTP requests, reading the response, and parsing it from JSON to Java objects, and makes it easy to reason about.

3.4.1 Factories

To hide the implementation of the different layers of the API calls, the application uses a factory pattern. A class that wants to create a certain type of adapter or connection class must do so via

a factory. The factory only promises to return an object implementing a certain interface. Some classes may implement several of the interfaces that the factory supports, but this fact is not known outside of the package.



This decouples the implementation of making API requests from the parts of the systems using them. The api packages in the infrastructure and application layer only offers the rest of the system a number of interfaces, and a factory. The implementing classes are package private.

3.5 Identifying current bus

The task of keeping track of which vehicle the user is on is by a chain of events where the infrastructure and application layer do different parts of the task. This division of labor decouples the rest of the application from any knowledge of how the closest vehicle is found, and thus allows changing or extending the ways this can be performed in the future.

3.5.1 Infrastructure layer and wifi events

In the infrastructure layer the class WifiBroadcastReceiver listens for changes in the wifi state. When one is received, it sends an event to its listeners that either the wifi state has changed (has been turned on or off) or that there are new wifis available nearby. It does so by sending WifiStateChangedEvents and NewWifiScanAvailableEvents. Both these events pass along the relevant data: a boolean representing the state and a list of nearby BSSIDs, respectively.

3.5.2 Application layer and current vehicle events

The class NearbyBusScanner receives the wifi events from the infrastructure layer and uses these to locate any nearby buses, by checking for matches between the nearby BSSIDs and a list of BSSIDs for buses. If there is a match, it notifies its listeners. If the scanner is told that Wifi is turned off or on, it notifies it listeners of whether it can perform scans or not. Note however that these listeners are not aware of how the NearbyBusScanner figures out if there are vehicles nearby or how it knows if it can scan or not. The knowledge that the application uses Wifi to detects buses is completely isolated to this class. If other ways to locate nearby buses where implemented in the future, the rest of the application would not be aware of this fact.

The Manager object listens for changes reported by the NearbyBusScanner. It uses these events to update what vehicle the user is on by creating domain objects, making calls to APIs as necessary, and notifies its listeners once this is done. If the manager is told that scans can't be performed, it notifies its listeners of this.

3.5.3 Presentation layer

In the presentation layer, the activity for viewing, deleting and sending flags get notified by the manager of changes, and tells the user.

Note that there is, on the surface, some knowledge of the fact that the application uses wifi to determine what vehicles are nearby. However, this is only for presenting the user with this fact. The user interfaces writes out that the user should activate wifi to find nearby vehicles, but programmatically, the view is ignorant of why scans can't be performed. It activates scanning by telling the Manager to perform scans, and the manager forwards to the request to the NearbyBusScanner, which turns the wifi on. The Manager is unaware of how the NearbyBusScanner finds buses, as are the views.

3.6 Persistent data storage

The only data that is stored persistently are the flags. This is done by sending requests to our server. For more information about how they are stored, see https://github.com/Oscmage/DAT255_server/tree/master/docs.

4 Issues

Being in a prototyping stage, there are some security considerations to take into account when using and viewing the application. These should ideally be addressed, but are deemed to not pose any major security risks. A fully deployed project should address these issues.

4.1 API authorization

The authorization for the Västtrafik and ElectriCity API are hardcoded into Java files, and shared in the git repository. The strings are declared in a separate config class, so it would be an easy step to replace them with placeholder strings and to let each team member supply their own authorization.

4.2 Lack of testing

There are automated tests for the entire domain package, but not that many for the rest of the code. Features and a polished graphical user interface has been prioritized before test driven development up until now. There are a document of acceptance tests, but these have to be stepped through manually. Writing more automatic tests of all types should be a priority in the future.

References

1. Layered Architecture: Evans, E. (2004). Domain-Driven Design: Tackling Complexity in the Heart of Software. Prentice Hall.
2. Design patterns: Gamma, E., Helm, R., Johnson, R. och Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.