Politecnico di Milano
AA 2017-2018

Computer Science & Engineering
Software Engineering Report

Travelendar+

Design Document
Requirement Analysis and Specification Document

Mingju Li - 10574864/898045
Chang Lin - 10597034/894651

Description of the problem

Many endeavors require scheduling meetings at various locations all across a city or a region (e.g., around Lombardy), whether for work or personal reasons (e.g., meeting the CEO of a partner company, going to the gym, taking children to practice, etc.). The goal of

this project is to create a calendar-based application that: (i) **automatically computes and accounts for travel time between appointments to make sure that the user is not late for appointments**; and (ii) **supports the user in his/her travels**, for example by **identifying the best mobility option** (e.g., use the train from A to B and then the metro to C), **buying public transportation tickets**, or by **locating the nearest bike of a bike sharing system**.

Users can **create meetings**, and **when meetings are created at locations that are unreachable in the allotted time, a warning is created**. As mentioned, the application should also **suggest travel means depending on the appointment** *(e.g., perhaps you bike to the office in the morning, but the bus is a better choice between a pair of afternoon meetings, and a car – either personal, or of a car-sharing system – is best to take children to practice)* **and the day** (*e.g., the app should suggest that you leave your home via car in the morning because meetings during a strike day will not be doable via public transportation; it could also take into account the weather forecast, and avoid biking during rainy days*).

Travlendar+ should allow users to **define various kinds of user preferences**. It should **support a multitude of travel means**, *including walking, biking (own or shared), public transportation (including taxis), driving (own or shared), etc.* A particular user may **globally activate or deactivate each travel means** (*e.g., a user who cannot drive would deactivate driving).* A user should also be able to **provide reasonable constraints on different travel means** *(e.g., walking distances should be less than a given distance, or public transportation should not be used after a given time of day).* Users should also be able, if they wish to, to **select combinations of transportation means that minimize carbon footprint**.

Additional features could also be envisioned, for instance **allowing a user to specify a flexible "lunch"**. *For instance, a user could be able to specify that lunch must be possible every day between 11:30- 2:30, and it must be at least half an hour long, but the specific timing is flexible. The app would then be sure to reserve at least 30 minutes for lunch each day. Similarly, other types of breaks might be scheduled in a customizable way.*

1. Introduction

1.1 Purpose

The purpose of this document is to give technical details advices about the architecture on the based of Requirement Analysis and Specification Document(RASD) about Travelendar+ (From here on, to make it brief we call it **TVLD** in abbreviation).

As a Design Document, This document is addressed to developers who must implement the requirements and could be used as a contractual basis and aims to identify:

- The high-level architecture and boundaries of system
- The design patterns
- The workflow of the information system
- The main components and their interfaces provided one for another
- The potential riskes and flaw
- The Runtime behavior

1.2 Scope

TVLD is designed to offer the user supports on the arrangements of timetable. TVLD is an information system based on mobile application and web application, which integrates information from different external systems and offer a highly optimized view of the information it obtains, using algorithms to calculate the best solution for a current situation.

It has one target group of users, people who would like to use this App to manage their time schedule. Of course, this group of users includes people involving different needs of time schedule managements, for example, a business manager may have different need from a housewife or a single student. Here are some basic classification don't in the market analysis:

People with Different Economic Ability
People with Different Time Flexibility
People with Different Attention on the Environment
……

Besides the user, external companies are the potential targets of this App. To build this system we need the information of different external systems like (ATM, MoBike, …), and this stimulates the increase use of these third-party service as well. Thus, we could make profits based on this win-win strategy.

Generally speaking, this system will automatically compute the travel time between different appointments to help users to well manage their schedules, and provide necessary supports for them. If the location is reachable in the allotted time, the system will provide the most time-economic or cost-economic travel means depending on the user choice. Otherwise, if

the location is unreachable, a warning will be alerted. The system will also take accounts of any other issues like weather, traffic jam or public strike. Ultimately, Travlendar+ should provide functions easy to use and allow user to set their own preference.

1.3 Definitions, Acronyms, Abbreviations

RASD: The Requirement Analysis and Specification Document
DD: Design Document
API: Application Programming Interface
Push Notification: a notification sent by a smartphone
UML: Uniform Model Language

1.4 Reference Documents

Teaching material from the Software Engineering 2 Professor Di Nitto
Specification Document: “Mandatory Project Assignments 2017 - 2018.pdf”.
GPS Performances: “http://www.gps.gov/systems/gps/performance/accuracy/".
UML Guidance: “http://www.html.it/guide/guida-uml/”.
StarUML Official Document: “http://docs.staruml.io/en/latest/”
Alloy Dynamic Model example: “http://homepage.cs.uiowa.edu/~tinelli/classes/181/Spring10/Notes/09-dynamic-models.pdf"
IEEE Std 830-1993 - IEEE Guide to Software Requirements Specifications.
IEEE Std 830-1998 - IEEE Recommended Practice for Software Requirements Specifications.

1.5 Document Overview

Introduction: this section inherit the part of RASD. It contains a review of all the requirements and functions we have to realize. Besides these, there is also an appendix and a complete description of the structure of the document

Architecture Design:

1. Overview: this section intends to demonstrate the whole relationships and roles they played in the whole system.
2. High level components: this section intends to show some integrated components and their functions in the system.
3. Components (in a relatively low level): this section intends to give a more detailed view of the components implemented in this system.
4. Deploying view: this section shows the part which needs to be deployed in previous. And the hardware/software pre-conditions if we need our system work.
5. Runtime view: this section shows the real workflow and work sequence of the components when performing different tasks in the real use.
6. Components interfaces: this section interfaces between different components (high levels or low levels)

7. Architectural styles and patterns: this section explain the engineering choice and decision we took during the design and programming of the application
8. Other supplement details

Algorithm Design: This section shows the core algorithms which Is going to be implemented in the system. According to our RASD, we need to design a algorithm integrates different traffic routes, computes time cost, and select one best route. This could be realize by Breadth first search, depth first search or A star search. In this part a short discussion of the advantages and disadvantages would be performed, to compare and reach a practical solution. The final result algorithm would be given in forms of pseudo code/java/python

User Interface Design: this section presents the graphical user interface on the smartphone (here we took iOS platform as an example) and explained by UML.

Requirement Traceability: This section confirm how the technical decisions taken link between RASD and DD.

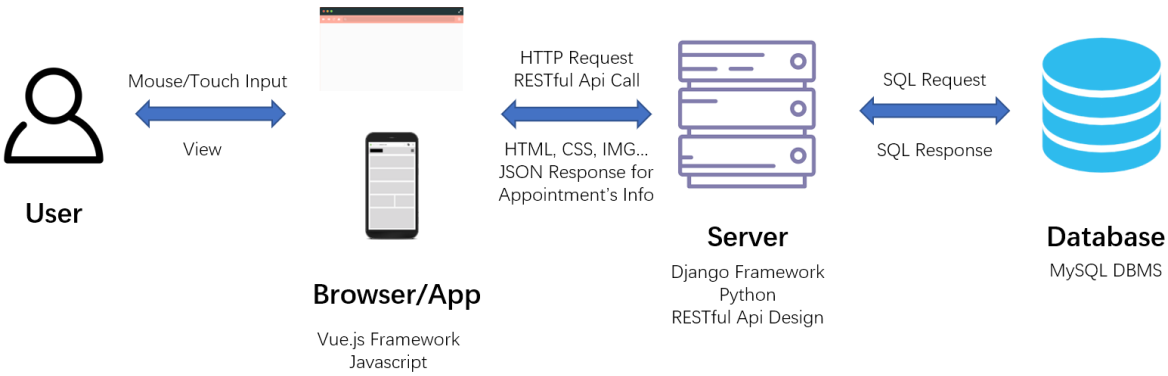
Implementation, integration and test Plan: This section identifies here the order in which our plan to implement the subcomponents of out TVLD system and the order in which we plan to integrate such subcomponents and test the integration.

Effort spent and reference: This section the information about the number of hours each member has worked for this document

2. Architectural Design

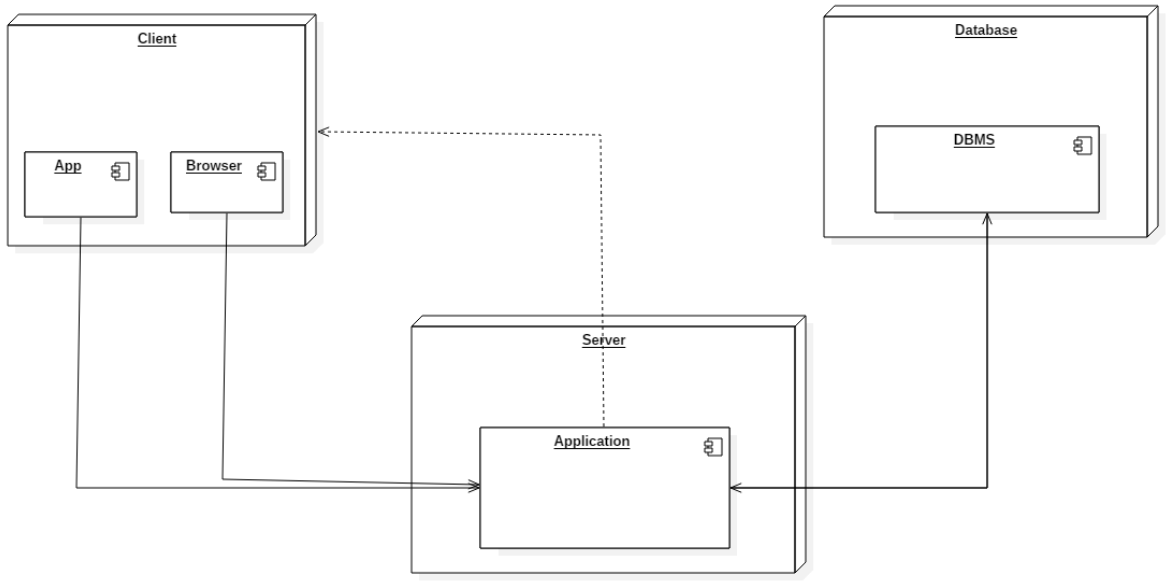
2.1 Overview

The general architecture is a tier architecture. It contains the client (Browser/App), the server and the database.



It's a linear architecture, which means the data flow between the user and the database is direct and bidirectional. No matter the user is using browser or App to user TVLD, the data binding to the user id will be operated by the same server and stored into the same database. This guarantees the synchronization of the system, which is also the premise for a cloud server.

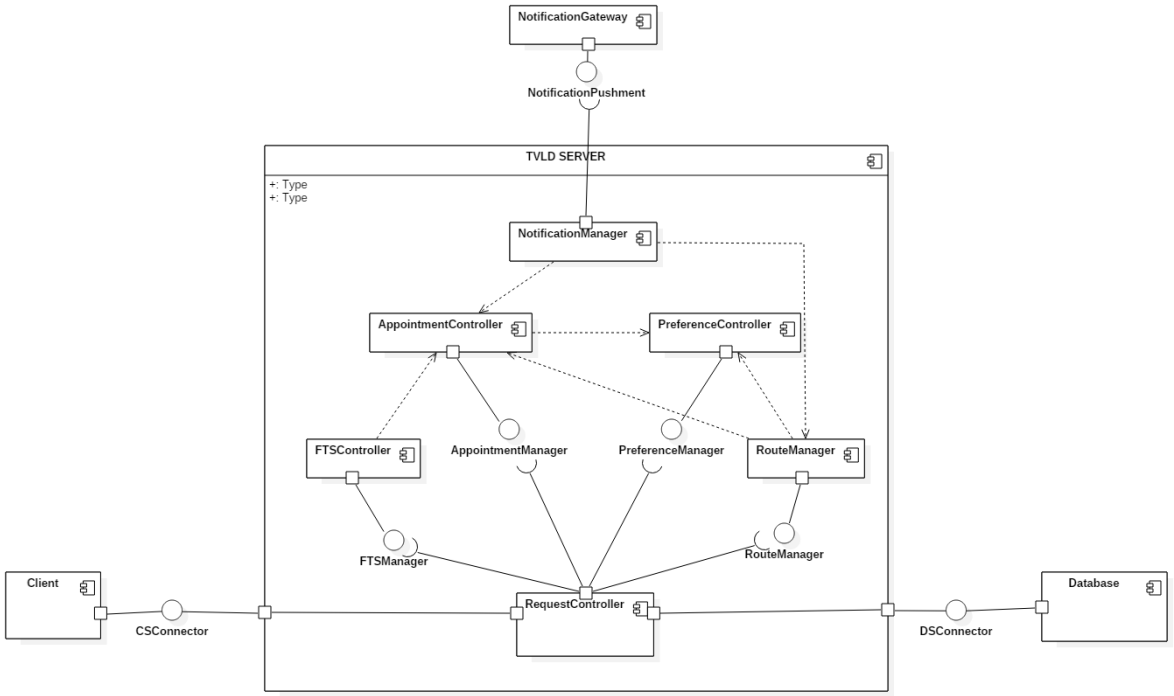
2.2 High-level Component and their interaction



The high-level components is like the general architecture. It contains three major elements. The first element is the most important part, a singleton server. The server can receive HTTP request from the client and then start a synchronous connection between the server and the client. The server will then send sql message to DBMS to retrieve data according to the client's request and send these data back to the client in JSON format. Also whiling the request contains the request for travel means of the appointment, the server will send message of appointment's position to an external map system to retrieve basic travel means data and do the analysis on the server based on user reference. After that, the server will send an asynchronous message back to the client, providing the best travel mean as the analysis result. The second element is the client part. It's also the only interface between the user and the system. It can receive input from the user and then send a HTTP request to the server. Then after receiving data from the server, the client will render the data in App/Browser to the user.

The third element is the database, which stores all the information of the appointments for each user. It has the only interaction with the server. After receiving the sql request from the server, it can send two types of responses back. If the sql operation succeeds, the confirm message will be sent. If a conflict or an error exists, the fail state with the error detail will be contained in the message.

2.3 Component View



- RequestController: manage the request
- FTSController: manage the free time slot
- AppointmentController: manage the appointment of the user
- NotificationManager: manager the notification to send
- PreferenceController: manage user's preference
- RouteManager: manage the route of the appointment
- TVLD Server: the main server of the TVLD system
- Client: user's device (browser or App)
- Database: the database to store all user's data
- NotificationGateway: manage the notification pushed to the user

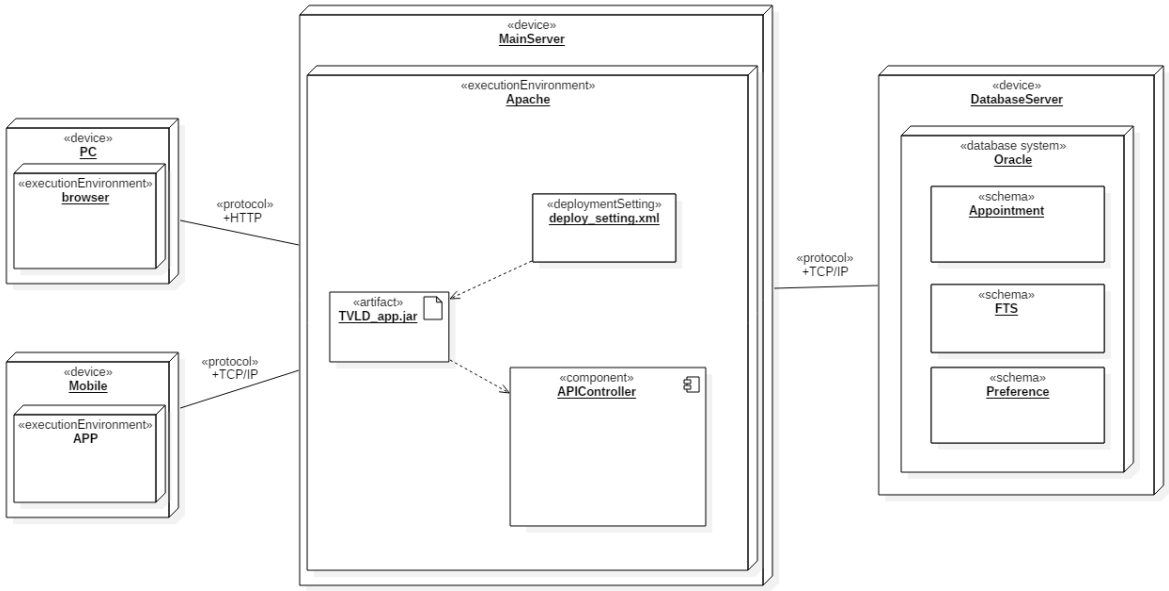
The system adopts a RESTFul Api for interaction between each elements (Server, Client and Database), to guarantee the synchronization of the data flow. For the components inside the server, the system adopts inner messages communication.

The request controller will dispatch the request from other architecture elements (Client and Database) and push the request to different components according to the header. Also, it will manage the request generated from the server and send them to different elements.

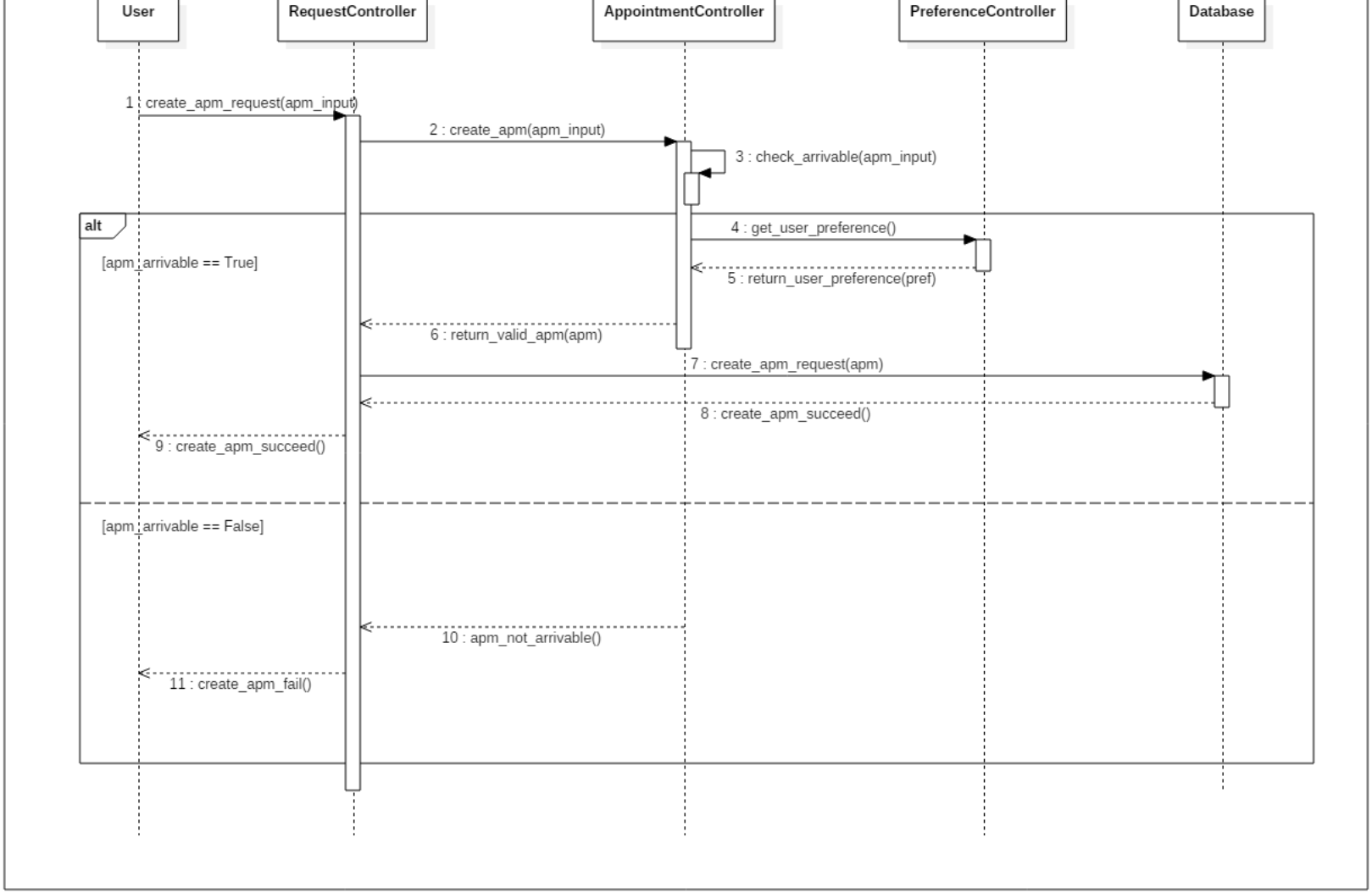
RouteManager manages all the route information with the user's appointment. The basic route will be retrieved from an external map system, which means the current position and the position of the appointment will be sent to the external map system via its interface.

NotificationManager will interact with the NotificationGateway via the interface. All notification pushed to the user should be done here.

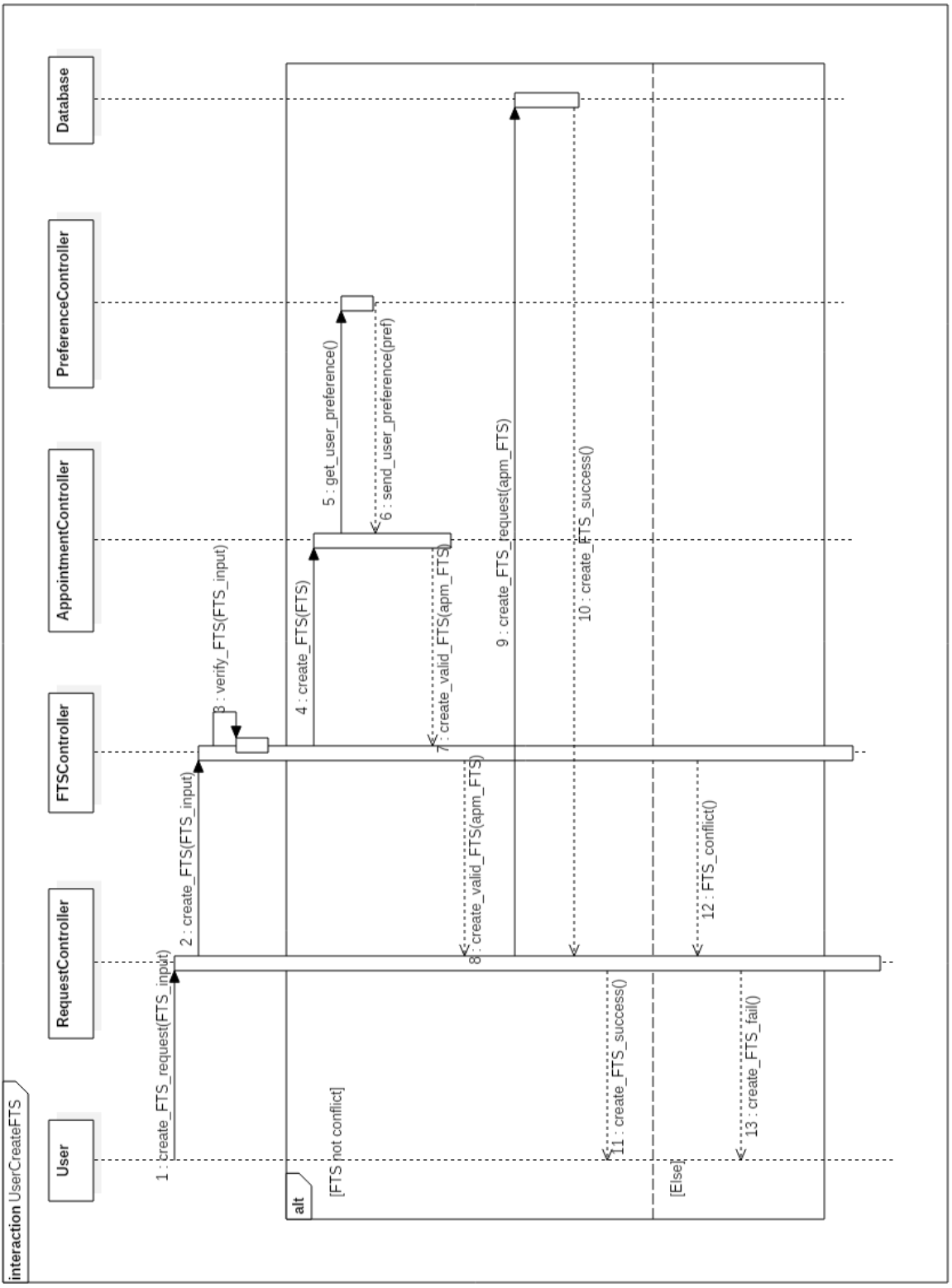
2.4 Deployment View



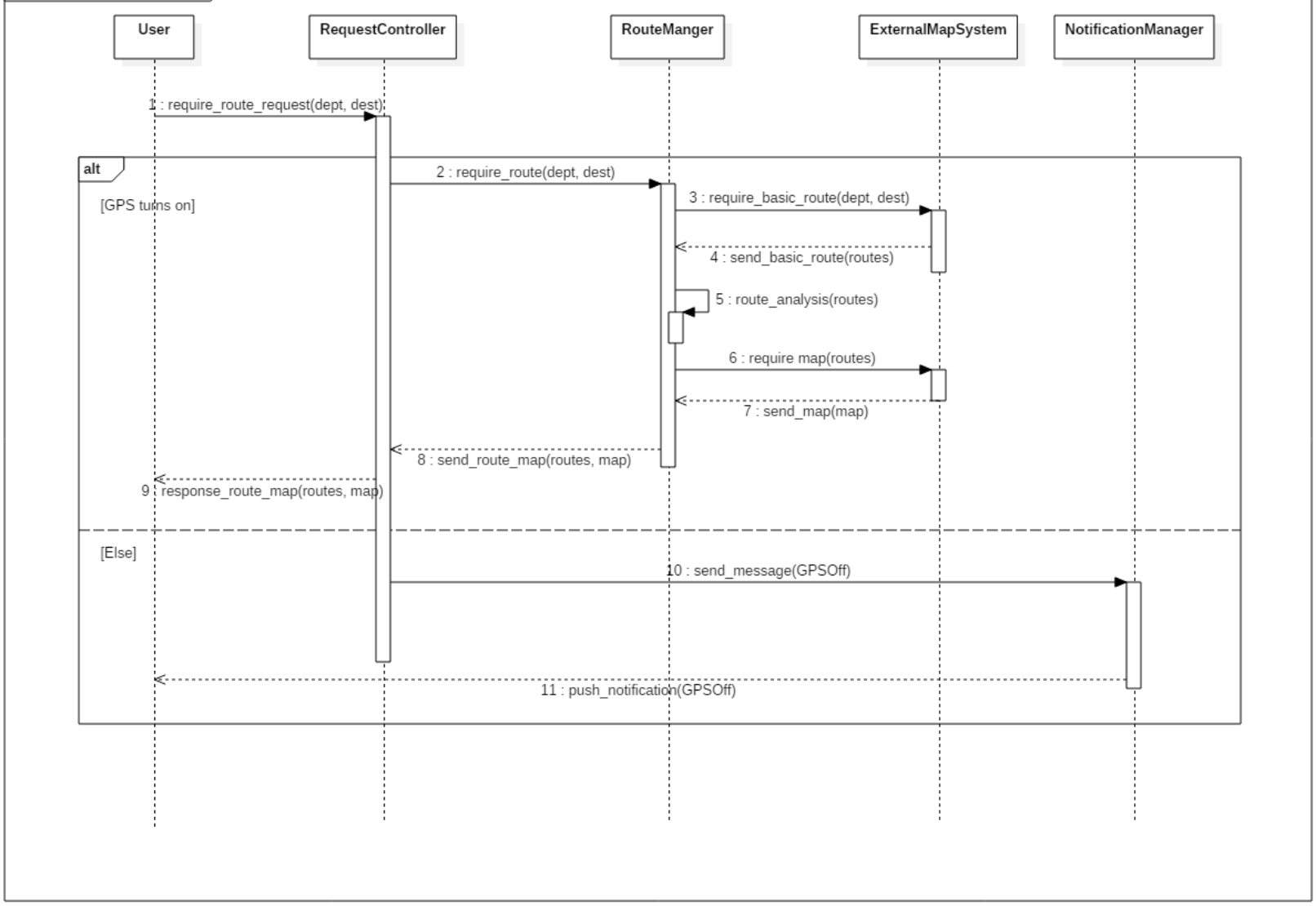
2.5 Runtime View


User
RequestController
AppointmentController
PreferenceController
Database
1 : create_apm_request(apm_input)
2 : create_apm(apm_input)
3 : check_arrivable(apm_input)
alt
[apm_arrivable == True]
4 : get_user_preference()
5 : return_user_preference(pref)
6 : return_valid_apm(apm)
7 : create_apm_request(apm)
8 : create_apm_succeed()
9 : create_apm_succeed()
[apm_arrivable == False]
10 : apm_not_arrivable()
11 : create_apm_fail()

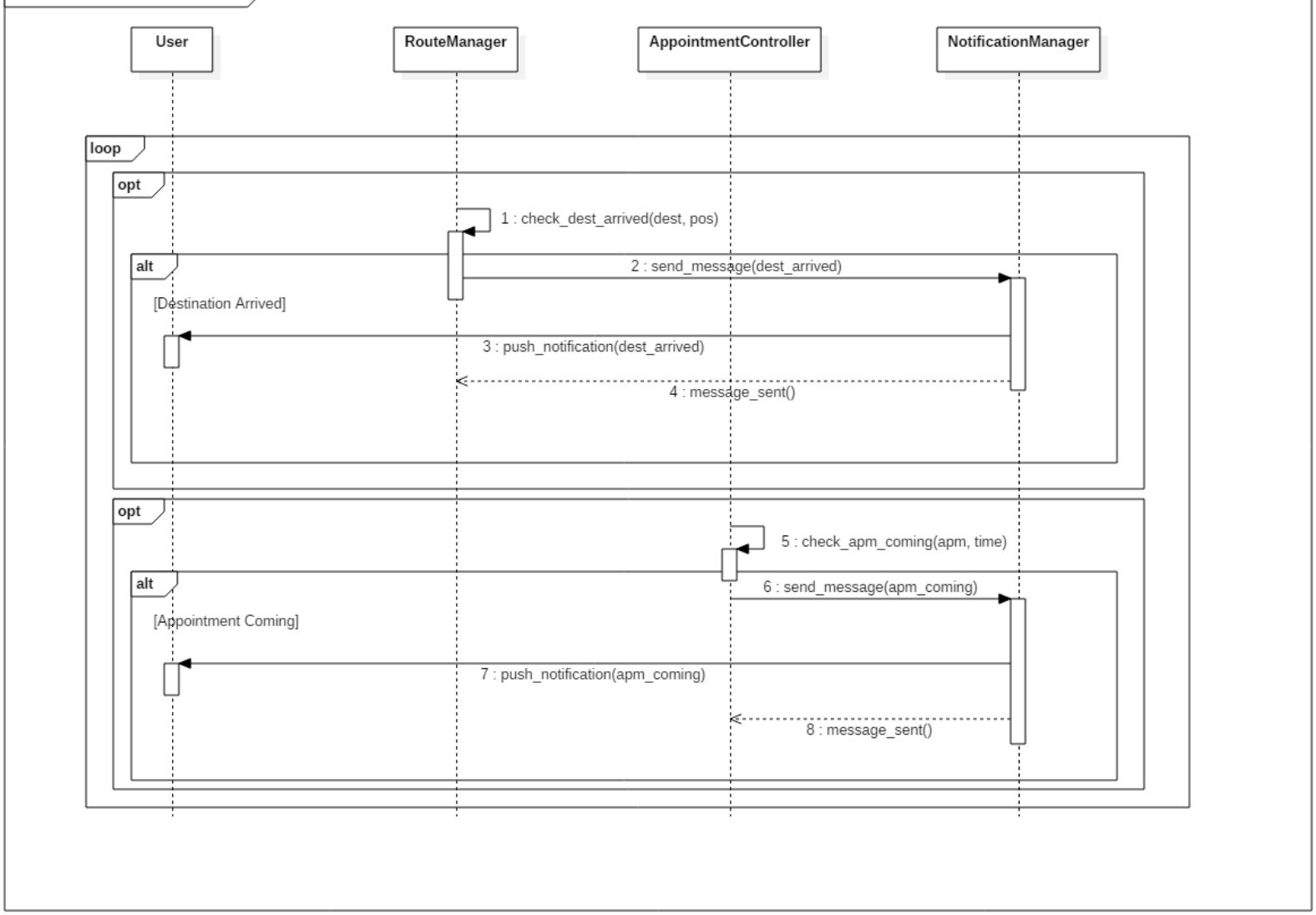
In the runtime view it can seen that once the user want to create an appointment, a request containing the appointment's position and time will be sent to the RequestController. The controller will then check the request's header and then forward it to the AppointmentController. Appointment's detail will be checked here and verifies the input is valid or not. If the position cannot be reached in time, a warning message should be sent back to the user. If valid, the AppointmentController will sent a request to Database to store all information about the appointment.


interaction UserCreateFTS
User
RequestController
FTSController
AppointmentController
PreferenceController
Database
1 : create_FTS_request(FTS_input)
2 : create_FTS(FTS_input)
3 : verify_FTS(FTS_input)
4 : create_FTS(FTS)
5 : get_user_preference()
6 : send_user_preference(pref)
7 : create_valid_FTS(apm_FTS)
8 : create_valid_FTS(apm_FTS)
9 : create_FTS_request(apm_FTS)
10 : create_FTS_success()
11 : create_FTS_success()
12 : FTS_conflict()
13 : create_FTS_fail()
alt
[FTS not conflict]
[Else]

In the runtime view it can be seen that the sequence creating a FTS has something in common with creating an appointment. Here FTS can be considered as a special appointment but is more flexible. Once the user tries to create a FTS, a request will also be sent the RequestController and forwarded by it according to the request's header. The FTSController will check its effectiveness and then send the details to the AppointmentController to create FTS (special appointment). Then also a request will be sent to the Database to store the FTS.

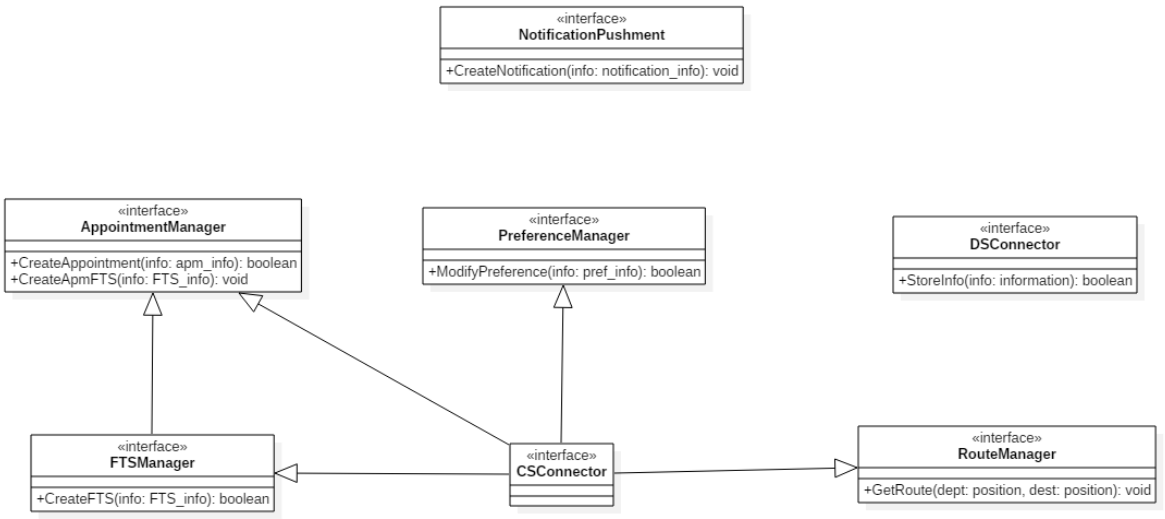

User
RequestController
RouteManger
ExternalMapSystem
NotificationManager
1 : require_route_request(dept, dest)
alt
[GPS turns on]
2 : require_route(dept, dest)
3 : require_basic_route(dept, dest)
4 : send_basic_route(routes)
5 : route_analysis(routes)
6 : require map(routes)
7 : send_map(map)
8 : send_route_map(routes, map)
9 : response_route_map(routes, map)
[Else]
10 : send_message(GPSOff)
11 : push_notification(GPSOff)

In the runtime view it can be seen that once the user want to follow the routine of an appointment, a request will be sent to the RequestController and then be forwarded to the RouteManager. The RouteManager will firstly communicate with an external map system to retrieve basic routes and the map contains that region. Then an advanced analysis will be done here to filter the basic routes according to user's preference and improve the quality of the route. After that, the final route will be sent to the user. All these process rely on the GPS function of the user's device. If the GPS isn't turned on, a warning message will be sent to the user to inform that.


User
RouteManager
AppointmentController
NotificationManager
loop
opt
1 : check_dest_arrived(dest, pos)
alt
2 : send_message(dest_arrived)
[Destination Arrived]
3 : push_notification(dest_arrived)
4 : message_sent()
opt
5 : check_apm_coming(apm, time)
alt
6 : send_message(apm_coming)
[Appointment Coming]
7 : push_notification(apm_coming)
8 : message_sent()

In the runtime view it can be seen that the notification function is realized by the NotificationManager. Here it’s mainly responsible for two types of notification. One is when the user arrives the destination of the appointment, a notification should be sent to the user to inform that. The other case is to inform the user for the coming appointment. All these are done in a loop to keep track on the user’s position and the time.

2.6 Component Interfaces



2.7 Architectural Styles and Patterns

TVLD adopts a MVC pattern, which is Model-View-Controller pattern. In Application/Browser part, MVC is adopted in the Vue.js framework. In Server part, MVC pattern is also used here. The models are binding to the data inside the database. These models will be handled by the Controller, which is each components in server system, and the corresponding views will be sent to the user. MVC pattern is adopted for the following reasons:

- Following MVC help isolate and contain bugs occurrences and testing, while giving a proper architecture to maintain your code.
- Maintain the layer independence gives you the ability to swap one model implementation for another, with no hustle, it ensures improving, tweaking one layer has no impact on others.
- Enables the full control over the rendered HTML
- Provides clean separation of concerns (SoC)

Also, TVLD is based on a Client-Server model. The client part is the user's device (Application/Browser), while the server part is the main server. The model guarantees the sharing of data, which means data is retained by usual business processes and manipulated on a server is available for designated users (clients) over an authorized access. Also, every client is given the opportunity to access corporate information via desktop interface eliminating the necessity to log into a terminal mode or processor.

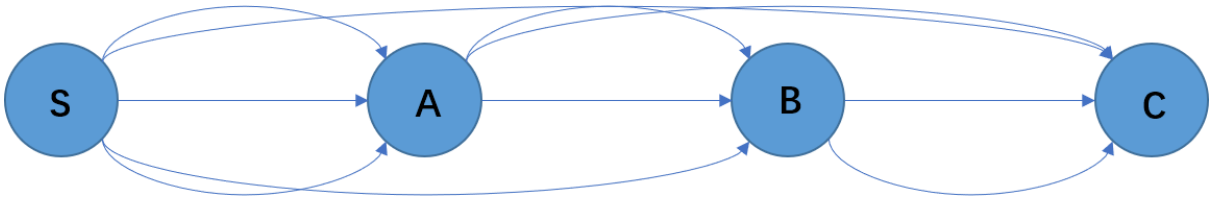
2.8 Other design decisions

TVLD should also integrates with an external map system, which requires a special interface for that.

3. Algorithms Design

Here we don't show all complete codes but just the critical ones.

In TVLD system, the most critical part is the analysis on the routes of the appointments. The RouteManager retrieves basic routes from external map system. Suppose the user has three appointment on that day, then the basic routes can be showed as following figure:



For each two nodes (position), several different routes (travel means) will be given by the external map system. Then the RouteManager should do the following analysis:

1. Filter routes based on user's preference. For example, if the user don't want to use bike, the route using a bike should be filtered.
2. Adopt A* algorithm and give the best route with less time/distance/… .

Pesudocode (A*):

```
function A*(start, goal)
    // The set of nodes already evaluated
    closedSet := {}

    // The set of currently discovered nodes that are not evaluated yet.
    // Initially, only the start node is known.
    openSet := {start}

    // For each node, which node it can most efficiently be reached from.
    // If a node can be reached from many nodes, cameFrom will eventually contain the
    // most efficient previous step.
    cameFrom := an empty map

    // For each node, the cost of getting from the start node to that node.
    gScore := map with default value of Infinity

    // The cost of going from start to start is zero.
```

```
    gScore[start] := 0

    // For each node, the total cost of getting from the start node to the goal
    // by passing by that node. That value is partly known, partly heuristic.
    fScore := map with default value of Infinity

    // For the first node, that value is completely heuristic.
    fScore[start] := heuristic_cost_estimate(start, goal)

    while openSet is not empty
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        closedSet.Add(current)

        for each neighbor of current
            if neighbor in closedSet
                continue            // Ignore the neighbor which is already evaluated.

            if neighbor not in openSet      // Discover a new node
                openSet.Add(neighbor)

            // The distance from start to a neighbor
            tentative_gScore := gScore[current] + dist_between(current, neighbor)
            if tentative_gScore >= gScore[neighbor]
                continue            // This is not a better path.

            // This path is the best until now. Record it!
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := gScore[neighbor] + heuristic_cost_estimate(neighbor,
goal)

    return failure

function reconstruct_path(cameFrom, current)
    total_path := [current]
    while current in cameFrom.Keys:
        current := cameFrom[current]
```

```
        total_path.append(current)
    return total_path
```

Java Code (A*):

```
public class AStar {
   // Amount of debug output 0,1,2
   private int verbose = 0;

   // The maximum number of completed nodes. After that number the algorithm
returns null.

   // If negative, the search will run until the goal node is found.
   private int maxSteps = -1;

   //number of search steps the AStar will perform before null is returned
   private int numSearchSteps;
   public ISearchNode bestNodeAfterSearch;

   public AStar() {

   }

   /**
   * Returns the shortest Path from a start node to an end node according to
   * the A* heuristics (h must not overestimate). initialNode and last found node
included.
   */

   public ArrayList<ISearchNode> shortestPath(ISearchNode initialNode, IGoalNode
goalNode) {
        //perform search and save the
        ISearchNode endNode = this.search(initialNode, goalNode);

        if(endNode == null)
            return null;

        //return shortest path according to AStar heuristics
        return AStar.path(endNode);
   }

   /**
   * @param initialNode start of the search
   * @param goalNode end of the search
   * @return goal node from which you can reconstruct the path
```

```
    */
    public ISearchNode search(ISearchNode initialNode, IGoalNode goalNode) {
        boolean implementsHash = initialNode.keyCode()!=null;
        IOpenSet openSet = new OpenSet(new SearchNodeComparator());
        openSet.add(initialNode);
        IClosedSet closedSet = implementsHash ? new ClosedSetHash(new
SearchNodeComparator())
                                : new ClosedSet(new SearchNodeComparator());
        // current iteration of the search
        this.numSearchSteps = 0;

        while(openSet.size() > 0 && (maxSteps < 0 || this.numSearchSteps <
maxSteps)) {
            //get element with the least sum of costs from the initial node
            //and heuristic costs to the goal
            ISearchNode currentNode = openSet.poll();

            //debug output according to verbose
            System.out.println((verbose>1 ? "Open set: " + openSet.toString() +
"\n" : "")
                        + (verbose>0 ? "Current node: "+currentNode.toString()+"\n":
"")
                        + (verbose>1 ? "Closed set: " + closedSet.toString() : ""));

            if(goalNode.inGoal(currentNode)) {
                //we know the shortest path to the goal node, done
                this.bestNodeAfterSearch = currentNode;
                return currentNode;
            }

            //get successor nodes
            ArrayList<ISearchNode> successorNodes = currentNode.getSuccessors();
            for(ISearchNode successorNode : successorNodes) {
                boolean inOpenSet;
                if(closedSet.contains(successorNode))
                    continue;


                /* Special rule for nodes that are generated within other nodes:
                * We need to ensure that we use the node and
                * its g value from the openSet if its already discovered
                */
                ISearchNode discSuccessorNode = openSet.getNode(successorNode);
                if(discSuccessorNode != null) {
                    successorNode = discSuccessorNode;
```

```
                    inOpenSet = true;
                } else {
                    inOpenSet = false;
                }

                //compute tentativeG
                double tentativeG = currentNode.g() + currentNode.c(successorNode);

                //node was already discovered and this path is worse than the last
one
                if(inOpenSet && tentativeG >= successorNode.g())
                    continue;

                successorNode.setParent(currentNode);

                if(inOpenSet) {
                    // if successorNode is already in data structure it has to be
inserted again to
                    // regain the order
                    openSet.remove(successorNode);
                    successorNode.setG(tentativeG);
                    openSet.add(successorNode);
                } else {
                    successorNode.setG(tentativeG);
                    openSet.add(successorNode);
                }
            }

            closedSet.add(currentNode);

            this.numSearchSteps += 1;
        }

        this.bestNodeAfterSearch = closedSet.min();

        return null;
    }

    /**
    * returns path from the earliest ancestor to the node in the argument
    * if the parents are set via AStar search, it will return the path found.
    * This is the shortest shortest path, if the heuristic h does not overestimate
    * the true remaining costs
    * @param node node from which the parents are to be found. Parents of the node
should
```

```
     *              have been properly set in preprocessing (f.e. AStar.search)
     * @return path to the node in the argument
     */
    public static ArrayList<ISearchNode> path(ISearchNode node) {

        ArrayList<ISearchNode> path = new ArrayList<ISearchNode>();
        path.add(node);
        ISearchNode currentNode = node;

        while(currentNode.getParent() != null) {
            ISearchNode parent = currentNode.getParent();
            path.add(0, parent);
            currentNode = parent;
        }

        return path;
    }

    public int numSearchSteps() {
        return this.numSearchSteps;
    }

    public ISearchNode bestNodeAfterSearch() {
        return this.bestNodeAfterSearch;
    }

    public void setMaxSteps(int maxSteps) {
        this.maxSteps = maxSteps;
    }

    static class SearchNodeComparator implements Comparator<ISearchNode> {
        public int compare(ISearchNode node1, ISearchNode node2) {
            return Double.compare(node1.f(), node2.f());
        }
    }
}
```

4. User Interface Design

4.1 Mock Up Demos

The mock up demos have been finished in the RASD.

4.2 UX Diagrams

We insert the UX Diagrams in UML to illustrate the relationship of different components and the main functions of the user.

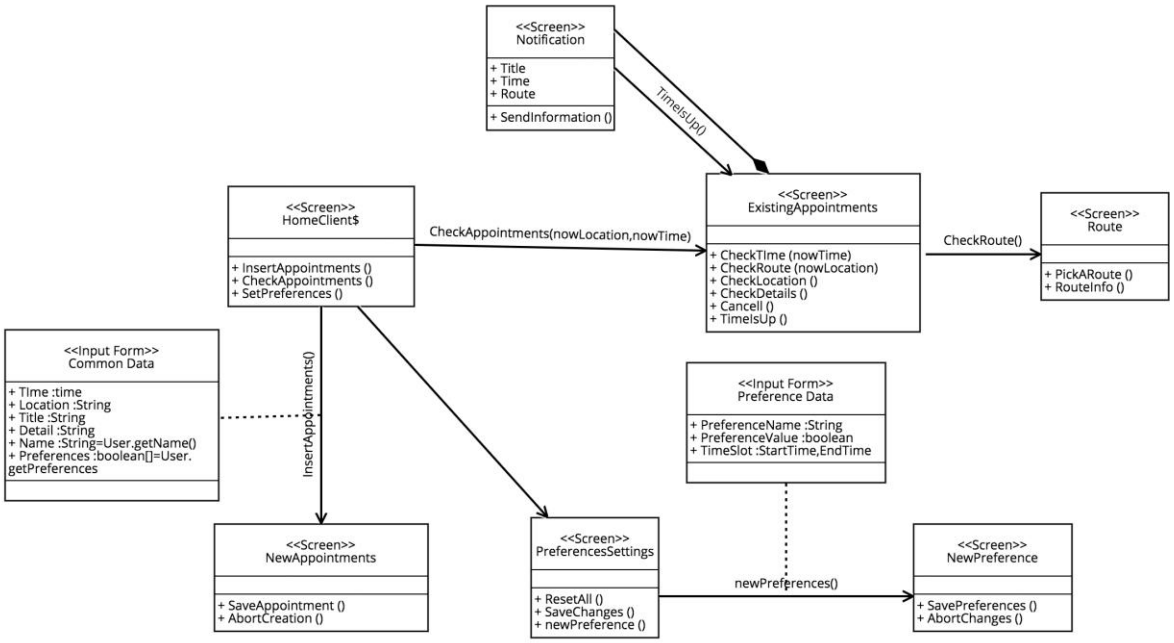


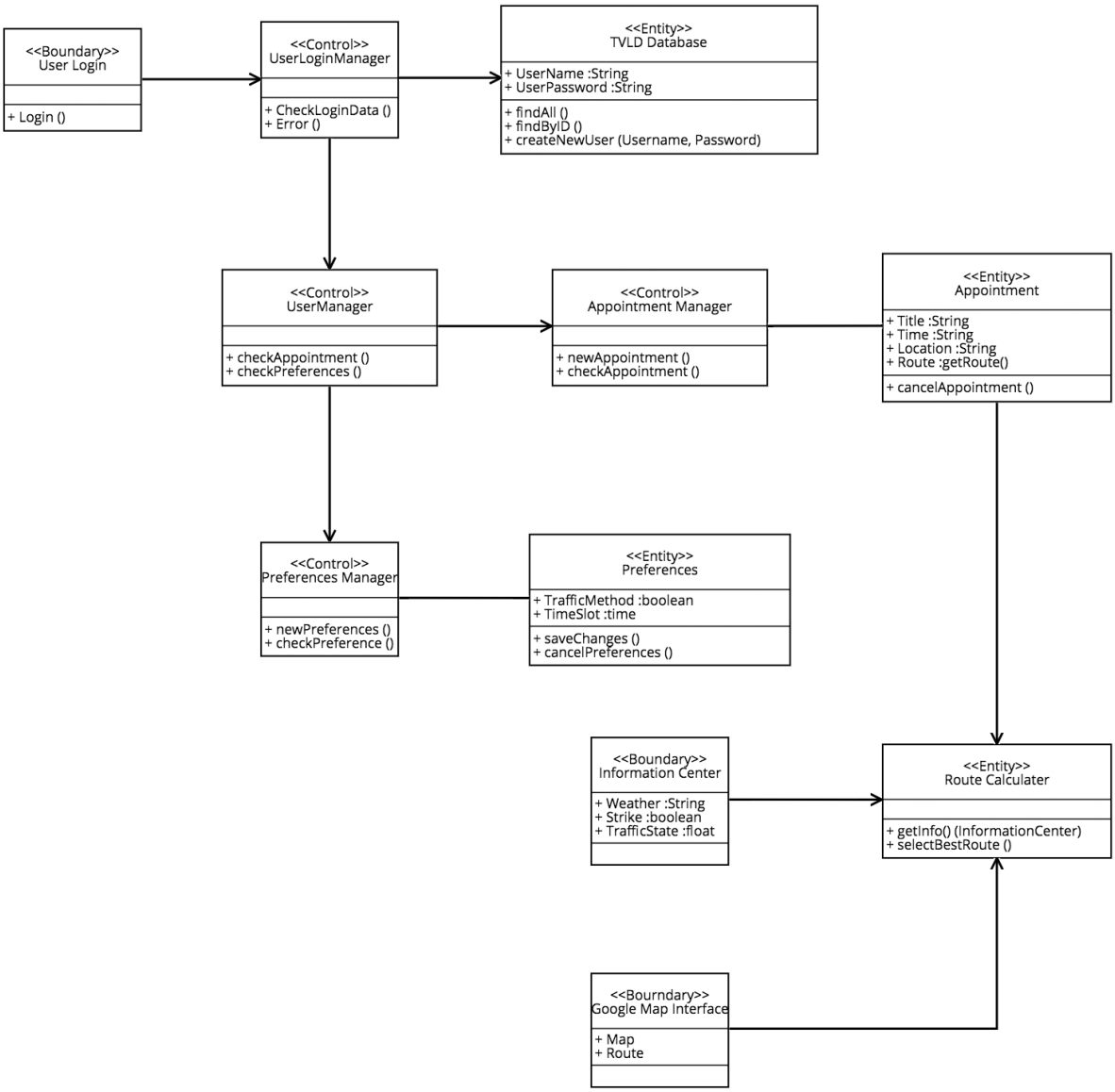
Figure UX User Mobile

4.3 BCE Diagrams

We insert BCE Diagrams to illustrate the functions of different and how they could be combined together to realize all the functions brought up in our RASD, and how they cooperate together to full fill all the requirements.

The Entities involved are following entity:

- RequestController: manage the request from all the mobile app
- UserLoginManager: manager the information inserted when the user login the APP
- FTSController: manage the free time slot
- AppointmentController: manage the appointment of the user
- NotificationManager: manager the notification to send
- PreferenceController: manage user's preference
- RouteManager: manage the route of the appointment
- TVLD Server: the main server of the TVLD system
- Client: user's device (browser or App)
- Database: the database to store all user's data
- InformationManager: the information of the situation of the route (weather and strike...)
- Map Interface: the interface of the external map information system from which TVLD obtain information and send the information to the route manager
- NotificationGateway: manage the notification pushed to the user



Comment: Here all the information managed by the FTS controller and Preference Manager are managed in the very same mode, so in this diagram the FTS controller is not specified.

5. Requirement Traceability

Here in the beginning of this part, to confirm that all the functions and goals illustrated in RASD have been implemented in our architecture, we list the RASD goals and their components related in the beginning part.

[G1] Allow users to set their time schedule including the location and time.
- -The UserLoginManager and its interface to database
- -The AppointmentManager Component

[G2] Allow the users to define their preference on the traffic method.
- -The UserManager
- -The PreferenceManager
- -The CloudServer*

[G3] Allow the users to define free time slot. (The user could set a specific time slot for lunch or personal affairs during a preset time slot).
- -The UserManager
- -The PreferenceManager
- -The CloudServer*

[G4] Show the users the time which is going to be needed from one place to the next appointment or place, based on the user's' customized traffic preferences, the current traffic state and weather.
- -The AppointmentManager Component
- -The InformationCenter Component
- -The RouteCalculater

[G5] Push notifications to the users in time to make sure they acknowledge the coming appointments and reach the destination in time.
- -The AppointmentManager Component
- -The RouteCalculater
- -The NotificationManager (The push notification gateway)

[G6] Allow the user to acquire the correct information about locations of bike sharing points, entrances of subways or bus station and so on.
- -The GoogleMap Interface
- -The RouteCalculater

[G7] Allow a user to activate cloud service and share meetings on different devices.
- -The CloudServer*
- -The Intelligent Analysis Agent*

[G8] Allow a user to active Big Data Analysis Service to retrieve better services from the system.
- -The data handling center*

(The service of Cloud Service big data service are the potential services we could offer to the user. While now it is not our first stage requirements. In the modelling of RASD like Sequence diagram, flow diagram we show the role they played in our information system. However in the design document, to give a practical solution covering all the basic functions, the CloudServer and big data server entities do not appear in our system.)

6. Implementation Integration and Test Plan

6.1 Description

This part is composed based on all the information included in the Requirement Analysis Specification Documents and previous Design Document. Integrating implementation and tests are designed in this part.

6.2 Subcomponent Implementation

All the elements to be integrated could be divided into 3 categories:
Front-end Components: TVLD mobile interfaces
Back-end Components: TVLD internal algorithm and calculation
External Components: all the components related to the TVLD while are provided by the external systems and the DBMS

There are several types of integration:

1. backend with external components (for example, TVLD relies on the external map information to give suggestions on the time and route for the next appointment)
2. frontend with backend components (for example, TVLD user mobile APP should be able to communicate with the backend components to consult the information about the the weather situation, the traffic situation as a parameter to be considered in the algorithm used in the path selection, and then the route and other notifications would be sent back to the user mobile side)

To be more specific, all the partial integration in this example are listed here:

Backend Components & External Components:
-Login Module: DBMS
-Route Selection: Route Calculater, GoogleMap Interface, Information Center
-Notification Sending: Appointment Manager Notification Managern (Notification Interface)

Frontend Components & Backend Components:
-Route Selection: Route Calculater, Preference Settings
-Communication Module: TVLD Server, User Manager
After the partial integration is completed, it would be possible to integrated the whole components as an unity.

6.3 Integration Testing Strategy

Given the whole information system, we use a bottom up strategy in the implementation. We start from the basic components and add components one step each time. Once all the single components are implemented and tested, we could finish by testing the subsystems which the previous components take part of. This gives us immediately response and feedback for the development phase.

Login Module: all the users must insert the right password and username to get access to the existed setting and appointments. It requires the connection with the DBMS.

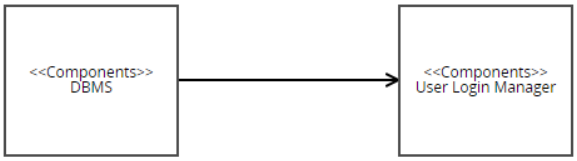


Figure-Login Module

Communication Module: whenever the user set a new appointment or modify his or her personal setting, the information would be save in the local memory, while at the same time all the information would be sent to the server of TLVD to calculate the best route and time evaluation.

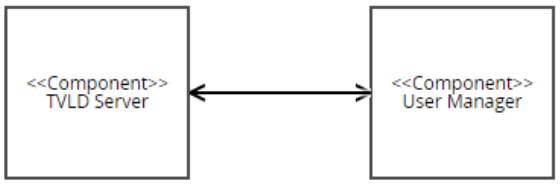


FIgure-Communication Module

Route Selection: all the route need to collect information from external components (like google map in this example), considering all the preference settings and information of the road like strike, weather condition

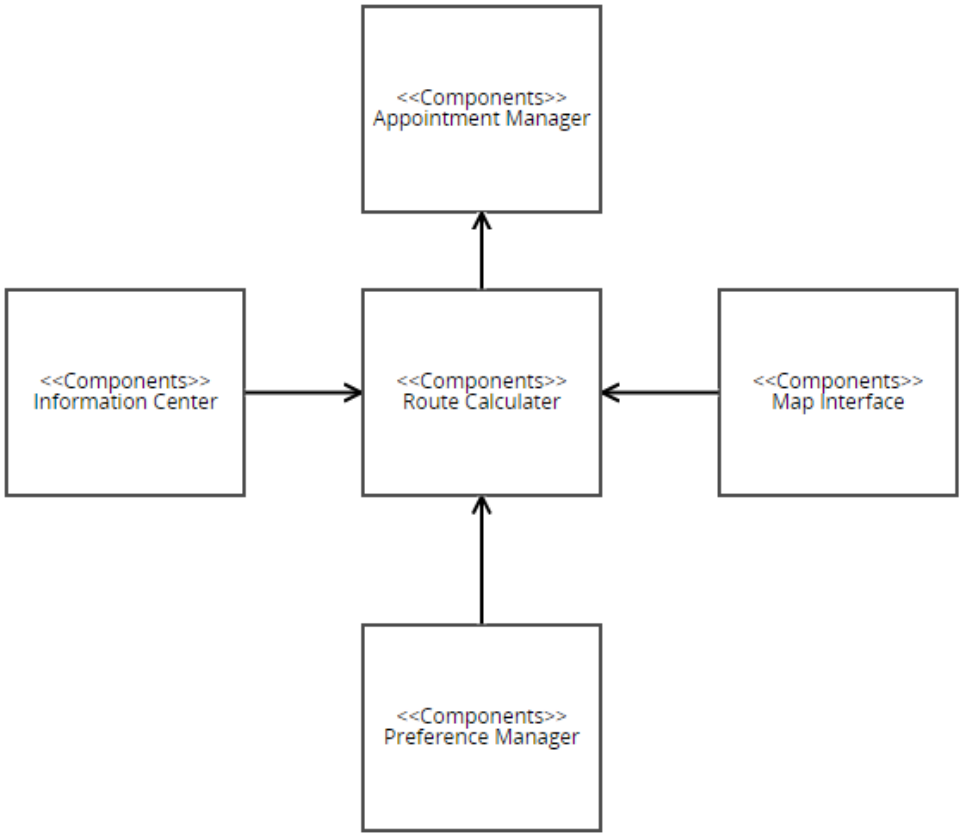


Figure-Route Selection

Notification Sending: when the application gather the information from the clock and the route time evaluation for each appointments, if the appointments are coming , the TVLD would push a notification to the user to remind him to start off to the next place.

Figure-Notification Sending

6.4 Individual Steps and Test Desciption

Here in this part, the steps of tests for each components (or subsystems) are demonstrated in detail. Noted that all these test examples are assumed that a proper database has been configured and activated, and the communication between the mobile application and relative interface has been configured rightly.

Login Module

Test ID	T1
Components	DBMS, User Login Manager
Input Specification	A registered user log in with a right password and username to get access to his previous preferences and appointments
Output Specification	Check if the user has a record in the database matching the information inserted in the login frame. Exception: if the user is new to the system, TVLD should be able to get the user registered into the system
Description	The data of the user are stored in the database of the server side. If the user get registered on another cell phone, he should be able to synchronize the data
Environment Needs	-

Create New Appointments

Test ID	T2
Components	DBMS, Appoint Manager
Input Specification	A user could insert [appointment location, title,description, time] to create a new appointment, and this would be also saved in the DBMS remotely
Output Specification	Check if the a new appointment could be created in the specific time and location. If correct, the data would be saved in the phone, and show the user “You have successfully created a new appointment” Exception: If another appointment has been created in an overlapping time, or this new appointment is contrast with the preset timeslot, if would remind the user that this appointment

	may fail to be executed and maybe need to rearrange the time.
Description	The user created a new appointment in his cellphone. TVLD could manage the player's appointments intelligently.
Environment Needs	-

Create/Modify Preference Settings

Test ID	T3
Components	Preference Manager, DBMS
Input Specification	A user could define his or her own preference settings. He could activate one method or disabled one method. Besides, he or she could create a flexible time slot of some length, to help the user better manage his timetable
Output Specification	Save the information
Description	The preference setting are saved locally in the cell phone. While some information are also synchronized in the server DBMS remotely. Exception: if the network is not good, which means that TVLD has to work offline, the preference setting would not be transferred to the Server and saved remotely. Instead it would saved locally and remind the user that the preferences would be synchronized when it is available to network
Environment Needs	-

Route Selection

Test ID	T4
Components	Preference Manager, Appointment Manager, Route Calculater, Map Interface, Information Manager
Input Specification	Each appointment would automatically calculate the route from the current position to the destination. This would require a map interface to offer it the solutions it has,the preference manager to filter the options, the information from the information center to help better evaluate the time which is going to be spent on the way
Output Specification	The time from the current position to the destination and the route..

Description	This is the core algorithm of the TVLD. It selects and shows route with given requirements on the route.
Environment Needs	T2,T3

Notification Sending

Test ID	T5
Components	Notification Center, Appointment Manager, Server
Input Specification	The real time of the world
Output Specification	If the time has come that the user need to prepare to move to next place, the notification would be sent to the TVLD Server, and then the Server make further routing steps.
Description	If any notification needs the user to move to the next destination, the notification center would send relative information to the main server, and then the server works to classify the things need to communicate
Environment Needs	T4

Communication Module

Test ID	T6
Components	TVLD Server, User Manager
Input Specification	the information of the appointment(location, time title, description), the FTS (begin time, end time), the preference setting
Output Specification	the route selection, the notification, the confirmation for creating a new appointment
Description	The user would be reminded if any of his appointment needs him to to move to the next destination
Environment Needs	T1,T2,T3,T4,T5

6.5 Test Example

Here we show parts of the test code for unit test and integration test.

```
public class LoginTester(){
   // Layer 1
```

```
    @Test
    public void testUsernameSize1(){
        String username = "Jack1995";
        assertTrue("UsernameSizeCheck works wrong with a correct username.",
usernameSizeCheck(username));
    }

    @Test
    public void testUsernameSize2(){
        String username = "Jack1995LoveMaria1996Forever";
        assertFalse("UsernameSizeCheck works wrong with a wrong username .",
usernameSizeCheck(username));
    }

    @Test
    public void testUsernameFormat1(){
        String username = "Jack1995";
        assertTrue("UsernameFormatCheck works wrong with a correct username.",
usernameFormatCheck(username));
    }

    @Test
    public void testUsernameFormat2(){
        String username = "Jack1995_+*&&^%$";
        assertFalse("UsernameFormatCheck works wrong with a wrong username.",
usernameFormatCheck(username));
    }

    @Test
    public void testPasswordSize1(){
        String password = "Maria1996";
        assertTrue("PasswordSizeCheck works wrong with a correct password",
passwordSizeCheck(password));
    }

    @Test
    public void testPasswordSize2(){
        String password = "Maria1996LoveJack1995Forever";
        assertFalse("PasswordSizeCheck works wrong with a wrong password",
passwordSizeCheck(password));
    }

    // Layer 2
    @Test(excepted = UsernameInputInvalidException.class)
    public void testUsernameInputInvalid1(){
```

```
        String username = "Jack1995LoveMaria1996Forever";
        InputUsername(username);
    }

    @Test(excepted = UsernameInputInvalidException.class)
    public void testUsernameInputInvalid2(){
        String username = "Jack1995_+*&&^%$";
        InputUsername(username);
    }

    @Test(excepted = PasswordInputInvalidException.class)
    public void testPasswordInputInvalid(){
        String password = "Maria1996LoveJack1995Foreve";
        InputPassword(password);
    }

    // Layer 3
    @Test
    public void testLoginSuccess(){
        String username = "Jack1995";
        String password = "Maria1996";
        assertTrue("Login fails with a correct username and password",
login(username, password));
    }

    @Test
    public void testLoginFail1(){
        String username = "Jack1995Wrong";
        String password = "Maria1996";
        assertFalse("Login succeeds with a wrong username", login(username,
password));
    }

    @Test
    public void testLoginFail2(){
        String username = "Jack1995";
        String password = "Maria1996Wrong";
        assertFalse("Login succeeds with a wrong password", login(username,
password));
    }

    @Test(excepted = NetworkDisconnectedException.class)
    public void testLoginWithNetworkConnection(){
        String username = "Jack1995";
        String password = "Maria1996";
```

```
        disconnect();
        login(username, password);
    }

    @Test(excepted = ServerBusyException.class)
    public void testLoginWithServerBusy(){
        String username = "Jack1995";
        String password = "Maria1996";
        for(long i=0; i<1000000; i++){
            login(username, password);
        }
    }
}

public class AppointmentTester(){
    @Test
    public void testCreateAppointmentSuccess(){
        Location loc = new Location("Via Roma, 8, Milano");
        String title = "Meeting";
        String description = "An important meeting with Prof. Roberto";
        Time time = new Time("16:00:00 10/11/2017");

        assertTrue("Create appointment fails with correct input.",
createAppointment(loc, title, description, time));
    }

    ...
}

public class PreferenceTester(){
    @Test
    public void testSetDriveForbidden(){
        setPreference(PreferenceEnum.DriveAllowed, false);

        assertFalse("Set drive forbidden fails.",
getPreference(PreferenceEnum.DriveAllowed));
    }

    ...
}

public class RouteSelectionTester(){
    @Test
    public void testSelectRouteWithDriveForbidden(){
        // Create an appointment
```

```
        Location loc = new Location("Via Roma, 8, Milano");
        String title = "Meeting";
        String description = "An important meeting with Prof. Roberto";
        Time time = new Time("16:00:00 10/11/2017");

        Appointment apm = createAppointment(loc, title, description, time);

        // Set preference
        setPreference(PreferenceEnum.DriveAllowed, false);

        // Select route
        for(Route route: getRoutes(apm)){
            for(Subroute subroute: route){
                assertEquals("Route selected doesn't match preference.",
TravelMeansType.Car, subroute.travelMean.type);
            }
        }
    }
}

public class NotificationTester(){
    ...
}

public class CommunicationTester(){
    ...
}
```

7. Effort Spent

Part/Hours Spent	Mingju Li	Change Lin
Introduction	2	1
Architectural Design	3	5
Algorithm Design	3	4
User Interface Design	2	1
Requirement Traceability	2	1
Implementation Integration	3	3
Total	15	15

8. References