Bilkent University

Department of Computer Engineering

Senior Design Project

Consigliere

Project Low Level Design Report

Selin Erdem, İrem Yüksel, Orhun Çağlayan, Furkan Küçükbay, Umut Mücahit Köksaldı

Supervisor: Assoc. Prof. Dr. Mehmet Koyutürk
Jury Members: Prof. Dr. Uğur Güdükbay
Prof. Dr. Cevdet Aykanat

Project Low Level Design Report

Feb 12, 2018

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Senior Design Project course CS492.

Table of Contents

1. Introduction

In the last few years, people's daily lives have changed in many different ways. Simple things became more complicated and we started to take more responsibilities and with more responsibilities, came higher expectations. Consequently, concepts such as making plans, using our time efficiently have gained more importance than ever before. Yet, it is a known fact that nowadays people struggle when it comes to planning a day and running errands. With the current technologies, it is possible to make a plan by entering the details such as time and place, and having reminder alerts for these tasks, but these applications do not help the user in terms of time efficiency and organization of their tasks.

Consigliere will be an iOS application that will work as a daily organizer and task manager. With Consigliere, we aim to help people use their time more efficiently and regain the time wasted on traffic. The user will simply need to enter whatever errands they have to run for that day and the application will provide an optimal plan for the user to complete these errands. This plan is designed by taking into account the roads the user will have to take to get to the locations of their tasks, and the traffic situation in the road. In addition, the application will estimate how much time the user will spend on an errand by analyzing the busyness of the location and the nature of the user's task. The application will also periodically check the traffic status of relevant roads and the crowd level of the locations in order to update the daily plan dynamically and send the user push notifications informing them of the opportunities to run their errands in a timely manner.

In this report, we aim to provide an overview of the low-level architecture and design of our system. First of all, the design trade-offs and the engineering standards are described. Also the documentation guidelines are listed. Afterwards, the packages in our system and their functionalities are described along with detailed class diagram views. Furthermore, interfaces of all classes in all packages are included. With descriptions the functionalities of each software component is clarified.

1.1 Design Trade Offs

1.1.1 Functionality vs. Usability

The main design goal of our system is providing the user with the maximum possible functionality with an easy-to-use interface as in most of the promising mobile apps today. Among the functionalities provided in Consigliere are authorization, task managing, optimization of daily plans, route planning according to several crucial constraints such as traffic intensity. These functionalities make Consigliere a mobile app that can be used on a daily basis and that can be employed by the user several times even in one single day. Therefore, user-friendliness is given the utmost importance to boost the fun the users are having while they are using Consigliere.

1.1.2 Security vs. Cost

Consigliere is an app that the user should, first, log in with their credentials. It is a platform where they are writing down their daily errands and also note personal reminders. In a way, Consigliere is the right hand of the user and as for all reliable servant, it should keep personal data for itself. In this regard, highly acknowledged Google Firebase is being employed for data needs.

1.1.3 Space vs. Time

As being an errand planner and daily plan optimizer, Consigliere requires a lot of space to both store the account details and the errands and the activity plans of the users. Real-time traffic data must also be handled for the app to optimize the daily plans. All these data needs however dictates requests from the app to the database and calls to the maps api. In the algorithmic level, slow processing time is going to be handled.

1.2 Engineering Standarts

In the reports, UML design principles are employed in diagrams, scenarios, use cases and class interfaces. Through all these, the system decomposition and the underlying structure of the system is described. The references in the reports follow the IEEE citation guidelines and thus are integrated into standards of the community.

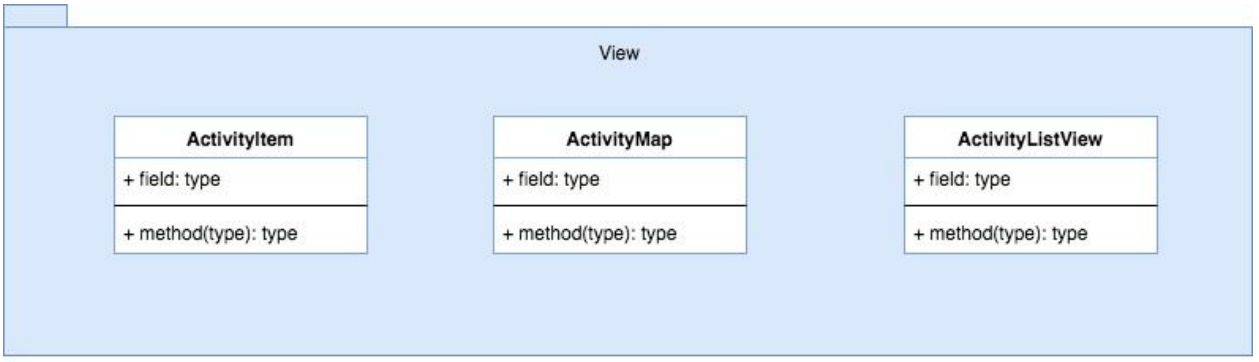
1.3 Interface Documentation Guidelines

In the implementation level, we follow the conventional ClassName format for naming our classes and functionName() format for naming the methods. A well-documented comments for classes and the methods are also present above the function and class declaration lines.

2. Packages

2.1 Client

2.1.1 View Package



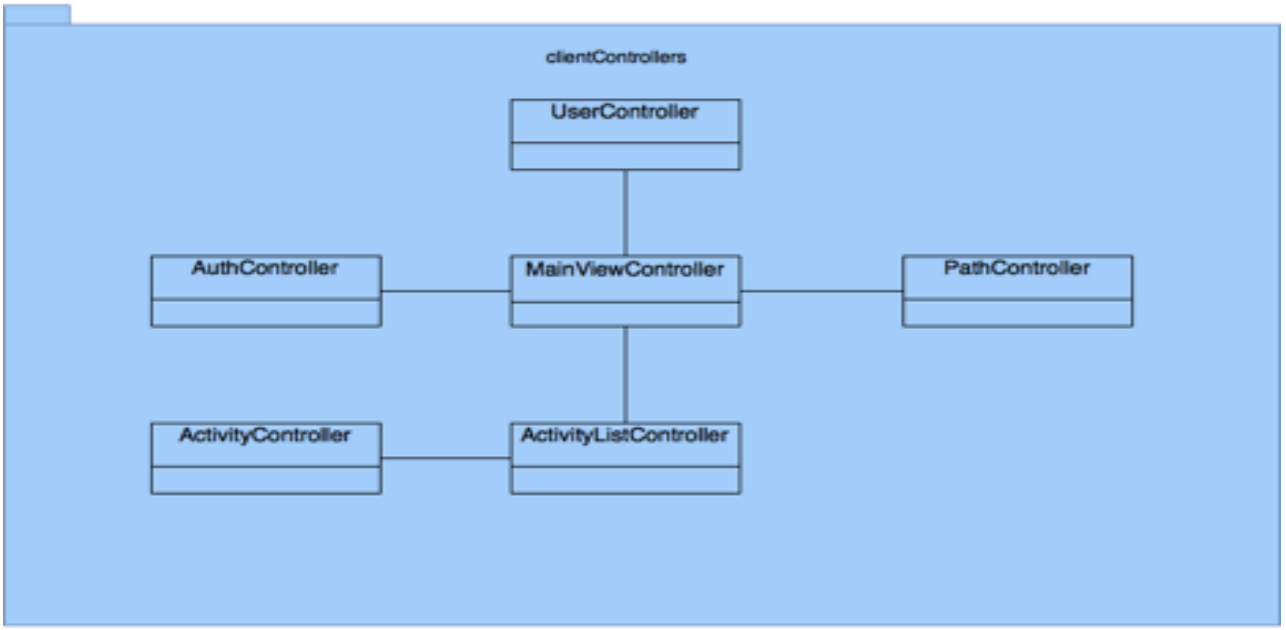
View package handles all user interface related operations.

ActivityItem: This class renders an activity list item for Activity.

ActivityMap: This class renders an interactive map for the current activity plan by drawing a path on the map. As the user scrolls through ActivityListView, the parent controller passes this ActivityMap a new Activity and this map transitions to show the path between two activity locations.

ActivityListView: This class renders a scrollable list for the plan of the current day. It also handles animations involving the list items and sends scroll events to the parent controller.

2.1.2 Controller Package



The controller package manages the interaction between user input, application level data model and server-side database.

MainViewController: This class is responsible for adjusting the components that are displayed on the screen according to the application level state.

AuthController: This class is responsible for the sign-up, sign-in and sign-out processes. If an account is to be created, it initializes the local user model in the system and stores the user data in the server. If the user logs in, then it retrieves the user data from the database and initializes the application level user model.

ActivityListController: This class is responsible for fetching and updating the current list of activities.

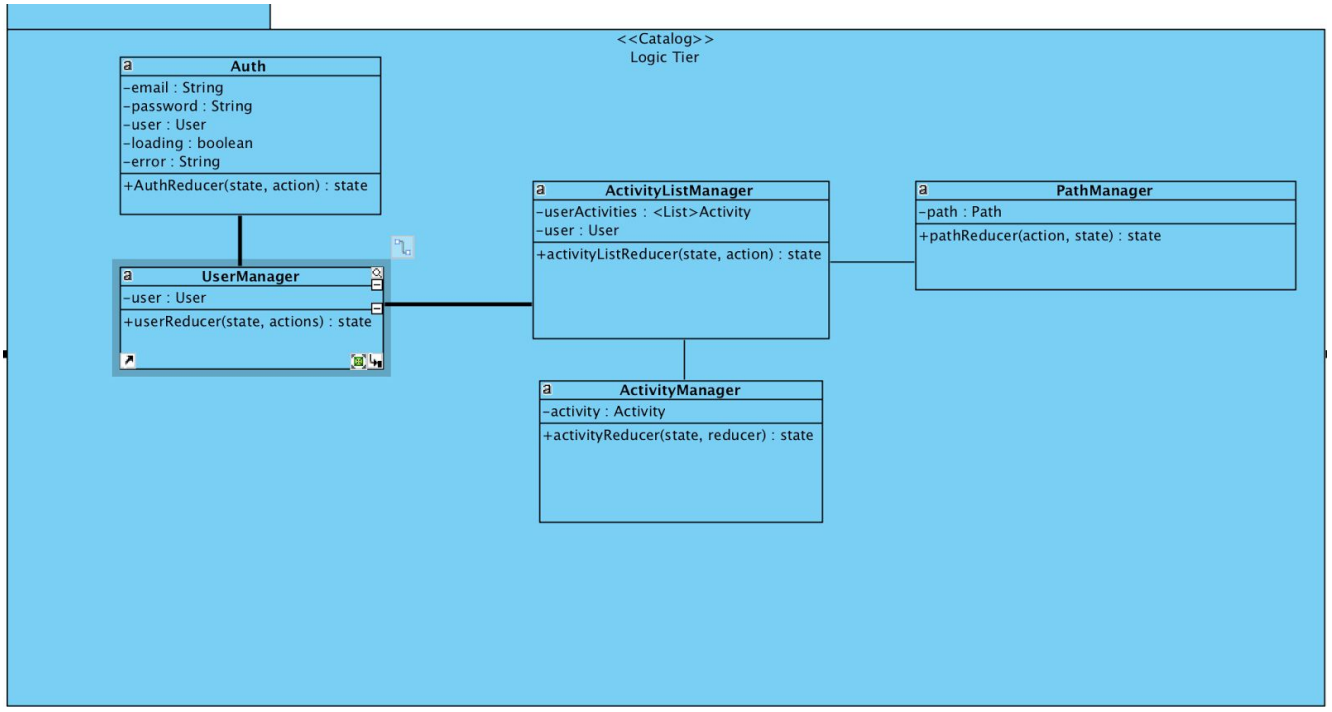
ActivityController: This class is responsible for the activity creation, update and delete procedures.

UserController: This class is responsible for setting the user preferences such as permanent addresses and on-going schedules.

PathController: This class is responsible for the calculation and the continuous update of the current optimized route for the given activities as well as updating the map display.

2.2 Server

2.2.1 Logic Package



Auth: This class is composed of authReducer and props that authReducer modify.

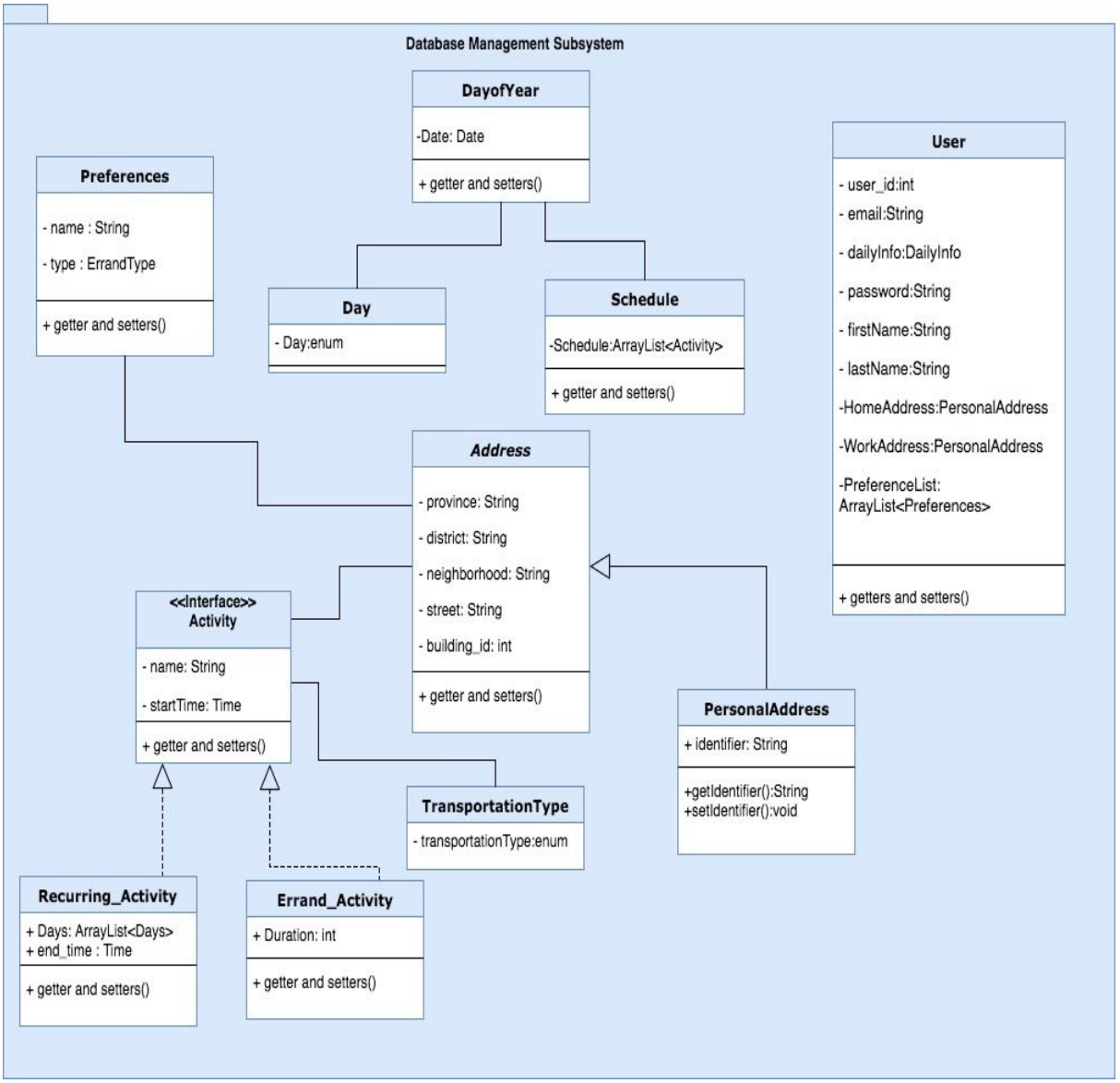
UserManager: This class is composed of userReducer and user prop, which handles modifications in user object (user profile).

ActivityListManager: This class is composed of userReducer and props that activityListReducer modify. This class help management of activities of user.

ActivityManager: This class is composed of userReducer and activity prop, which handles modifications in single activity.

PathManager: This class is composed of pathReducer and path prop, which manages path state.

2.2.2 Data Package



Schedule: This class represents the schedule that it created for the user which a list of activities.

DayofYear: This class represents a specific day of year. It holds the schedule and the date of a specific day.

Activity: This is an interface for Errand_Activity and Recurring_activity classes

Errand_Activity: This class represents the errands that user has to run in an irregular basis. It has duration attribute that helps the program to manage the schedule timely.

Recurring_Activity: This class represents the recurring activities of the user. More specifically, it represents the activities that a user do on a regular basis. It has days attribute that refers to the days that the activity recurs.

Day: This class represents the days of a week.

Preferences: This class represents the preferences of a user such as the locations that s/he prefer to go.

TransportationType: This class represents the possible transportation methods for an activity such as private car, walking and public transportation.

Address: This class represents the address information of an activity. The attributes corresponds to components of formal addresses in Turkey.

PersonalAddress: This class inherits Address class and represents the personal addresses of the user namely, home and work addresses.

User: This class represents a user of the application. A user object has identifying information such as name, e-mail address etc.

3. Class Interfaces

3.1 Client

3.1.1 Views

- **Class ActivityItem:** This class renders an activity list item for Activity.
  Properties:
  private Activity
- **Class ActivityMap:** This class renders an interactive map for the current activity plan by drawing a path on the map. As the user scrolls through ActivityListView, the parent controller passes this ActivityMap a new Activity and this map transitions to show the path between two activity locations.
  Properties:
  private ArrayList <Activity>
  Methods:
  public setActivityList(ArrayList <Activity>): Sets the current Activity list to the given Activity list.

- **Class ActivityListView:** This class renders a scrollable list for the plan of the current day. It also handles animations involving the list items and sends scroll events to the parent controller.
  Properties:
  private List <Activity>
  Methods:
  public setBlocks(List <Activity>): Sets the current Activity list to the given Activity list.

public addOnScrollListener(OnScrollListener listener): Adds a ScrollListener to the Activity List.
public addOnItemClickListener(OnItemClickListener listener): Adds a OnItemClickListener to each Activity on the Activity List.

3.1.2 Controllers

- **Class MainViewController:** This class manages which component will be rendered on the screen depending on where the user is currently browsing in the application.
  Properties:
  private Component activeView
  Methods:
  public LoginForm(): Switches the current view to the login form, adds any previous active views to the React Native view stack
  public ActivityList(): Switches the current view to the login form, adds any previous active views to the React Native view stack
  public ActivityDetails(): Switches the current view to the login form, adds any previous active views to the React Native view stack
  public MapView(): Switches the current view to the login form, adds any previous active views to the React Native view stack
  public UserPrefList(): Switches the current view to the login form, adds any previous active views to the React Native view stack

- **Class AuthController:** This class manages the authentication process for the users.
  Properties:
  private User currentSession
  Methods:
  public emailChanged(String): Is executed whenever a new character has been typed into the e-mail section on the login form, processes the newly entered string and calls AuthReducer to store the e-mail in application state
  public passwordChanged(): Is executed whenever a new character has been typed into the password section on the login form, processes the newly entered string and calls AuthReducer to store the e-mail in application state
  public userLogin():Tries to log the user in with the current credentials stored in auth state. Can call onLoginFail if there is no account associated with the given credentials. If the operation succeeds, retrieves the user data from the server and calls AuthReducer to store it in application state.
  public onLoginFail(): Tries to create an account with the given credentials, should that also fail, calls AuthReducer to set the error flag to 1.

- **Class UserController:** This class manages the maintenance and adjustment of the user's preferences such as their permanent schedule.
  Properties:
  private Date[] schedule
  Methods:
  public setDate(Date): Executed whenever the user fills out a previously empty time slot in their current schedule, the newly created date is added to the schedule array kept both in this class and in application level user preference state.
  public deleteDate(Date): Executed when the user clears out a previously filled slot in their schedule, finds and removes the given date object from the schedule object in user preference state.

- **Class PathController:** This class is responsible for the calculation and update of the current most optimized path that goes through all of the user's uncompleted activities.
  Properties:
  private Path[] currentPath
  private Activity[] currentActivities
  private float interval
  Methods:
  public calculatePath(): Executed upon user request, gathers location based data and calculates the most optimal path for the currently listed Activities
  public updatePath(): Executed at time intervals specified in the class properties.Checks the current activity locations and updates location data stored in the application state through the PathReducer. Then recalculates the most optimal path
  private pushActivities(Activity[]): Used by the updatePath and calculatePath methods to set the current Activity list on an application wide object.

- **Class ActivityListController:** This class is responsible for the addition and deletion of activities into the current list and its synchronization with the server.
  Properties:
  private Activity[] currentActivities
  Methods:
  public activityCreate(Activity): Called when a new activity is created, adds the newly created activity to the currentActivities array as well as storing it in the application level state using ActivityListReducer.
  public activityDelete(Activity):Called when an activity is deleted, removes the deleted activity from the currentActivities array as well as remving it from the application level state using ActivityListReducer.
  private fetchActivities(): Requests the user's stored activities from the online database.
  private pushActivities(): Called whenever a new activity is created or deleted, updates the activity snapshot stored in the database.

- **Class ActivityController:** This class manages the settings for an individual activity, whether during creation or editing.
  Properties:
  private String activityText
  private Location activityLocation
  private ActivityType activityType
  Methods:
  public activityTextChanged(String): Is executed whenever a new character has been typed into the activity description section on the activity details form, processes the newly entered string and calls ActivityReducer to store the text description in application state.
  public activityTextChanged(String): Is executed whenever a new character has been typed into the activity description section on the activity details form, processes the newly entered string and calls ActivityReducer to store the text description in application state.
  private fetchActivities(): Requests the user's stored activities from the online database.
  private pushActivities(): Called whenever a new activity is created or deleted, updates the activity snapshot stored in the database.

3.2 Server

3.2.1 Logic

- **Class Auth:** This class handles user authentication states and notifies other components about user authentication by modifying states and using props.
  Properties:
  private String email
  private String password
  private User user
  private String error
  private boolean loading
  Methods:
  public state authReducer(action, state)

- **Class UserManager:** This class is composed of userReducer and user prop, which handles modifications in user object (user profile).
  Properties:
  private User user
  Methods:
  public state authReducer(action, state)

- **Class ActivityListManager:** This class is composed of userReducer and props that activityListReducer modify. This class help management of activities of user.
  Properties:
  private User user
  private <List>Activity userActivities
  Methods:
  public state activityListReducer(action, state)

- **Class ActivityManager:** This class is composed of userReducer and activity prop, which handles modifications in single activity.
  Properties:
  private Activity activity
  Methods:
  public state activityReducer(action, state)

- **Class PathManager:** This class is composed of pathReducer and path prop, which manages path state.
  Properties:
  private Activity activity
  Methods:
  public state pathReducer(action, state)

3.2.2 Data

- **Class Schedule:** This class represents the schedule that it created for the user which a list of activities.
  Properties:
  private ArrayList<Activity>
  Methods:
  Getter and setter methods
  addActivity(Activity newactivity)

- **Class DayofYear:** This class represents a specific day of year. It holds the schedule and the date of a specific day.
  Properties:
  private ArrayList<Activity>
  private Date date
  private Day day
  private Schedule schedule
  Methods:
  Getter and setter methods

- **Interface Activity:** This is an interface for Errand_Activity and Recurring_activity classes.
  Properties:
  private String name
  private Time starttime
  Methods:
  Getter and setter methods

- **Class Errand_Activity implements Activity:** This class represents the errands that user has to run in an irregular basis. It has duration attribute that helps the program to manage the schedule timely.
  Properties:
  private int duration
  Methods:
  Getter and setter methods

- **Class Recurring_Activity implements Activity:** This class represents the recurring activities of the user. More specifically, it represents the activities that a user do on a regular basis. It has days attribute that refers to the days that the activity recurs.
  Properties:
  private ArrayList< Days>
  private Time end_time
  Methods:
  Getter and setter methods

- **Class Day:** This class represents the days of a week.
  Properties:
  public enum Day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}

- **Class Preferences:** This class represents the preferences of a user such as the locations that s/he prefer to go.
  Properties:
  private String name
  private ErrandType type
  Methods:
  Getter and setter methods

- **Class TransportationType:** This class represents the possible transportation methods for an activity such as private car, walking and public transportation.
  Properties:

public enum TransportationType { privateCar, walking,public_transportation, cycling}
Methods:
None

- **Class Address:** This class represents the address information of an activity. The attributes corresponds to components of formal addresses in Turkey.
  Properties:
  private String Province
  private String District
  private String Neighborhood
  private String Street
  private int Building_id
  Methods:
  Getter and setter methods

- **Class PersonalAddress inherits Address:** This class inherits Address class and represents the personal addresses of the user namely, home and work addresses.
  Properties:
  private String identifier
  Methods:
  Getter and setter methods

- **Class User:** This class represents a user of the application. A user object has identifying information such as name, e-mail address etc.

  Properties:
  private int user_id
  private String Email
  private String Password
  private String irstName
  private String lastName
  private PersonalAddress HomeAddress
  private PersonalAddress WorkAddress
  private ArrayList<Preferences> PreferenceList
  Methods:
  Getter and setter methods

4. References

[1]"Traveling Salesman Problem", Math.uwaterloo.ca, 2017. [Online]. Available: http://www.math.uwaterloo.ca/tsp/. [Accessed: 15- Oct- 2017].