Travlendar+
Design Document
Version 1.0

Leonardo Bisica
Alessandro Castellani
Michele Cataldo

Contents

1 Introduction

1.1 Purpose

The purpose of this Design Document is to explain the architecture of the Travlendar+ mobile application as well as the logic underlying its development. Our approach will be analyzed in detail section by section, keeping, above all, coherence with the path our RASD layed down.

1.2 Scope

Travlendar+ is a mobile application that encompasses many different functionalities. It is first of all an event scheduler that keeps track of a user's appointments so that it can display the fastest routes available, These routes are indexed by transportation means thanks to external APIs. *Travlendar+* heavily relies on "*Google Maps APIs*" for tracking distances and travel times, but also uses the necessary APIs to locate and rent vehicles of sharing services and to buy public transportation tickets. Thanks to its ability to gather external info about weather, strikes and traffic, as well as average travel time, *Travlendar+* can warn its users before creating an appointment (and during travel itself) if any overlap happens or if an event location can't be reached in the expected time.

1.3 Definitions, Acronyms, abbreviations

We assume our Glossary already cover all the terms we introduced and specified in the RASD. In addition to that, we can add some additional word to our vocabulary :
Abbreviations :

RASD The Requirements Analysis and Specifications Document is the first document we produced in order to lay the foundations of Travlendar+.

DD The Design Document is the document at hand.

API Application Programming Interface.

Travel Logic By travel logic we refer to the logic that processes the distances and the transportation time within our operative and influcence zones. In the case at hand, in this first implementation, we're going to adopt as Travel Logic the Google Maps APIs.

User The user is the final customer of *Travlendar+*, the ones which uses the mobile application we detail.

RWn The n^{th} runtime view.

1.4 Revision History

- Component "API Server" has been removed in Deployment View

1.5 Reference Documents

- Specification Document: Mandatory Project Assignments, available in the BeeP page of the course.

1.6 Document Structure

The document is organized into 7 sections:

- Section 1 (Introduction): the section at hand. It provides the scope of our project and frames our DD.
- Section 2 (Architectural Design): this section details the architecture of *Travlendar+*. It contains component, deployment and runtime views.
- Section 3 (Algorithm Design): this section displays the most important algorithms used by the mobile application.
- Section 4 (User Interface Design): this section briefly refers to the User Interface developed in the RASD.
- Section 5 (Requirements Traceability): this section tracks the requirements detailed in the RASD into their corresponding design elements.
- Section 6 (Implementation, Integration and Test Plan): this section identifies the order in which we implement the various subcomponents of the system as well as the order we want to implement and test them in.
- Section 7 (References & Used Tools): this section accounts for the references of our project, and the tools we adopted in order to write down and deliver this document.

2 Architectural Design

2.1 Overview

As we anticipated, in this section we're going to give an array of views of our system, shaping it from many angles at once. We'll detail both the high-level components and the interfaces interleaved among them, delving then into deployment and runtime view. The architectural style we chose to adopt is a three-layered one. The decision to implement an external DB was born out of the need to provide a realiable and safe synchronization tool for our users and to store other kinds of personal information submitted through the app (like preferences and the like).

2.2 Component View

The Component Diagram shown below describes the logical components of the system we are to develop, from a very high-level description to a more detailed one. The diagram also shows a net of depencies, in accordance to UML 2.0.

2.2.1 High Level Component View

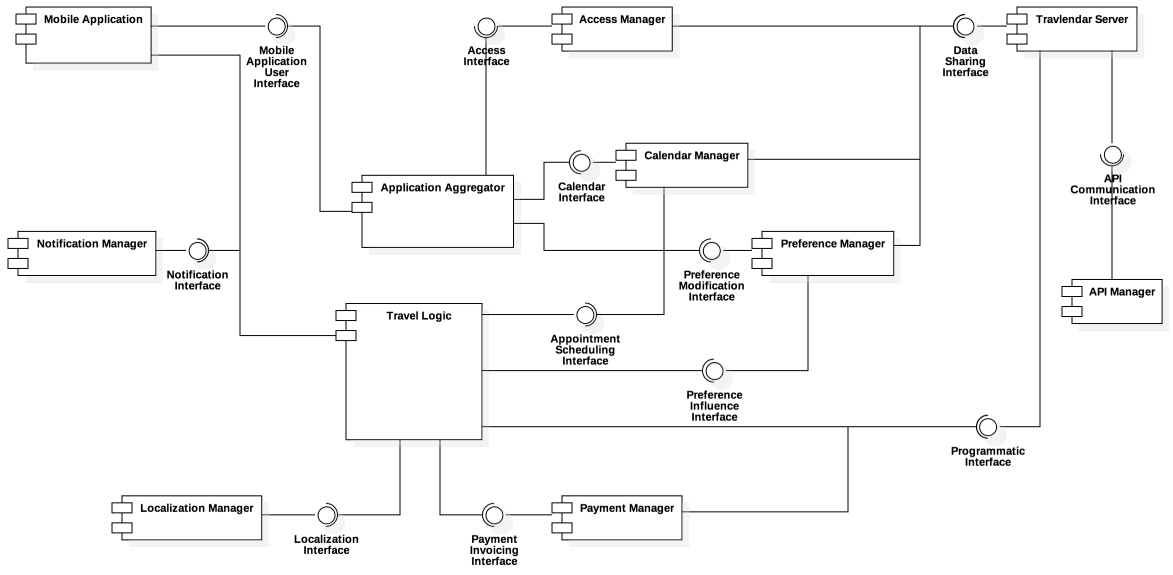


Figure 1: High level view.

We will now describe in detail all the components.

Mobile Application This component represents the view of the user over its system. It's split in two sub-components, Guest-view and User-view, which represent the two different ways a human interaction can be instaurated with the system.

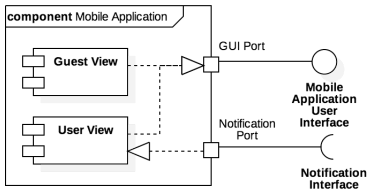


Figure 2: Mobile Application detail.

Access Manager This component deals with the matter of signing-up and authenticating to Travlendar+ Server. SignUp Handler serves the purpose of isolating the logic which serves the purpose to initiate a new User registration, while Saved Login Data records User data and smooths authentication procedures without contacting the external server. Authentication Manager oversees every access procedure.

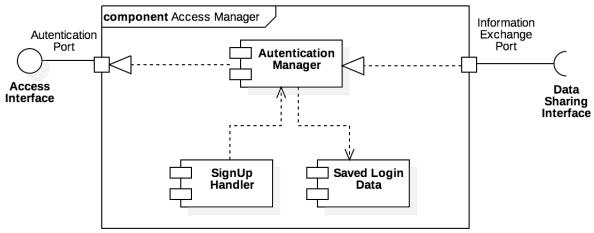


Figure 3: Access Manager detail.

Application Aggregator This component works, unsuprisingly, as a collector of the different information *Travlendar+* manages. It allows an easy management of every piece of information and allows us to avoid an high number of interfaces among the different components. 'Profile Manager' specifies the profile setting of the current user, while 'User Actions Handler' allows us to register user's input.

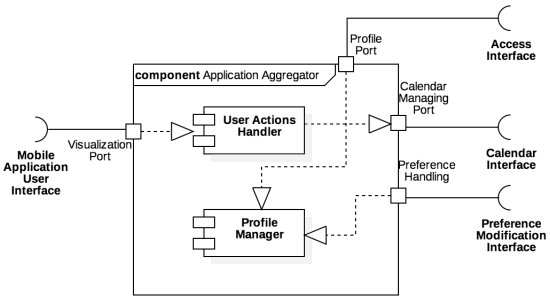


Figure 4: Application Aggregator detail.

Calendar Manager This component contains the Appointment Aggregator subcomponent , a sub-component that manages the appointments inserted by the user together with their properties (like date and location). The component also contains 'Trip List'

and 'Break List' sub-components : they deal with the storage and presentation of trips linked to appointments and break frames.

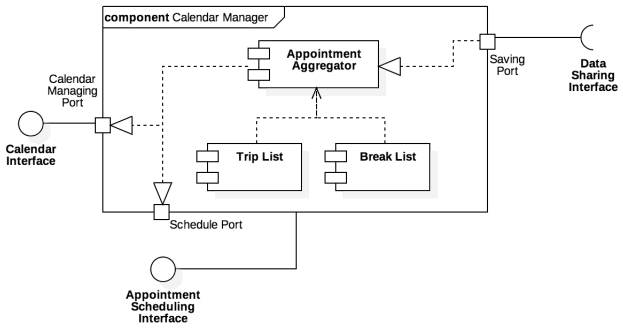


Figure 5: Calendar Manager detail.

Preference Manager This component serves the purpose of keeping track of the preferences expressed by the user. 'Season Pass Handler' takes care of storing season passes; 'Excluded Vehicles List' serves the purpose of cutting off from the scheduler results involving a selection of banned transportation means, 'Preferences List' covers the remaining and wider spectrum of User's choices. 'Preference Handler' is the sub-component that manages the other ones and that communicates outside Preference Manager.

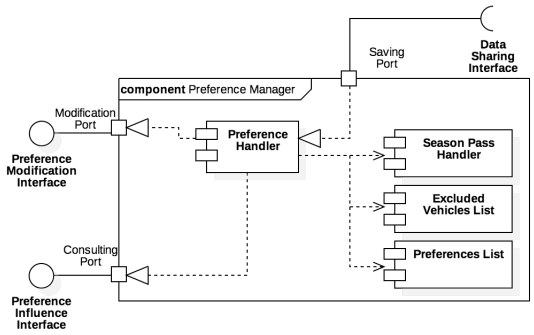


Figure 6: Preference Manager detail.

Travel Logic Manager This component is split into two sub-components: 'Scheduler' is the fundamental block that aims at scheduling and arranging user appointments and breaks via the the corresponding trips, 'Trip Handler' is the block whose purpose is to organize and present travel times to the scheduler in order to have them sorted out and well-managed.

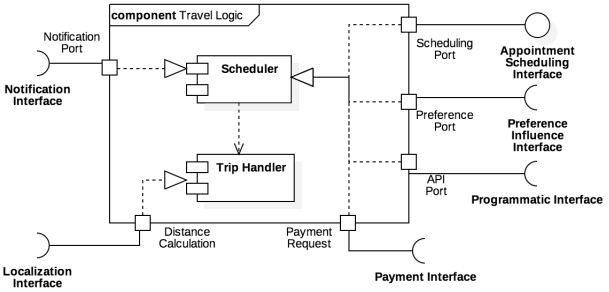


Figure 7: Travel Logic detail.

Payment Manager This component deals with the recording of purchases and their associated credit cards. The sub-component 'Payment Handler' tracks purchase records and interacts with the required apps installed on the mobile device, while 'Purchase History' is an exploitable and rational organizations of saved purchases and credit cards used by the user.

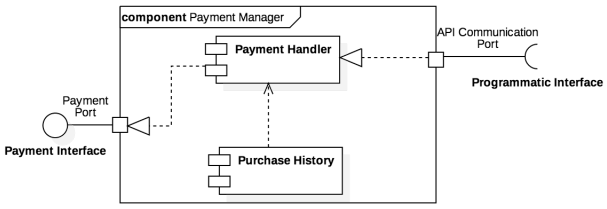


Figure 8: Payment Manager detail.

Travlendar Server This component represents the *Travlendar Server* whose purpose is to store user's preferences, access data and personal information. It is modeled by its 'DBMS'sub-component, which stores Timetables and Preferences and the 'API Request Dispatcher', which forwards requests to external agents.

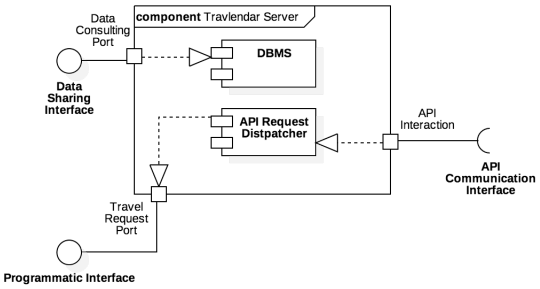


Figure 9: *Travlendar Server* detail.

API Manager This component is critical in order to provide a functioning Travel Logic: it gathers the interactions with all external APIs. Here are listed the APIs sub-components the mobile application project starts with : Google Maps API, Google Transit

API, Open Weather Map API, Car2Go and BikeMi API. 'Other API-Based System' subcomponent is here inserted for reference only, as previoulsy pointed out in the RASD. Naturally, the list of external services can be expanded. The 'Listener' component serves the purpose of forwarding requests : it receives the desired inputs and standardizes the formats and file types.

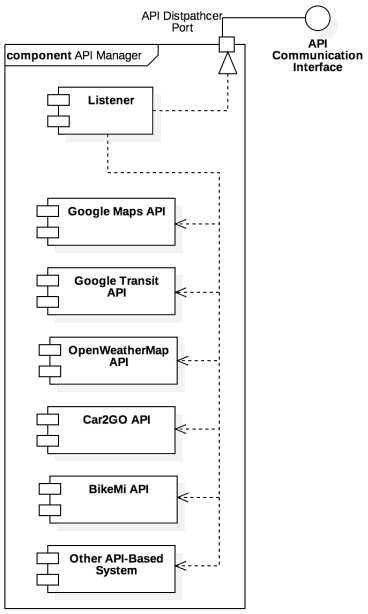


Figure 10: API Manager detail.

Notification Manager This component allows the notification system to warn users in case events partially or completely overlap according to the scheduler.

Localization Manager This component represents the localization functionalities of the mobile application.

2.2.2 Detailed Level Component View

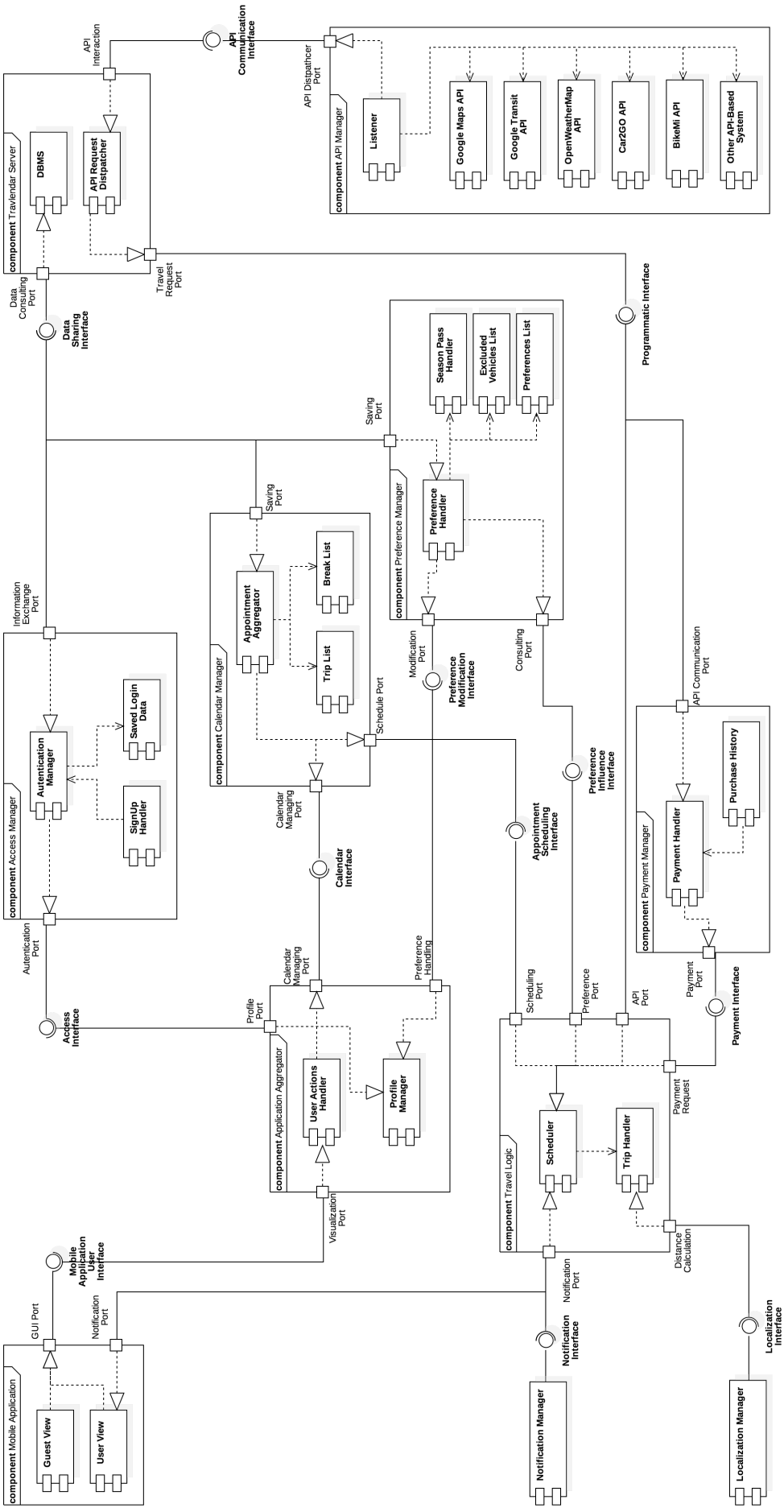


Figure 11: Detailed level view.

2.3 Deployment View

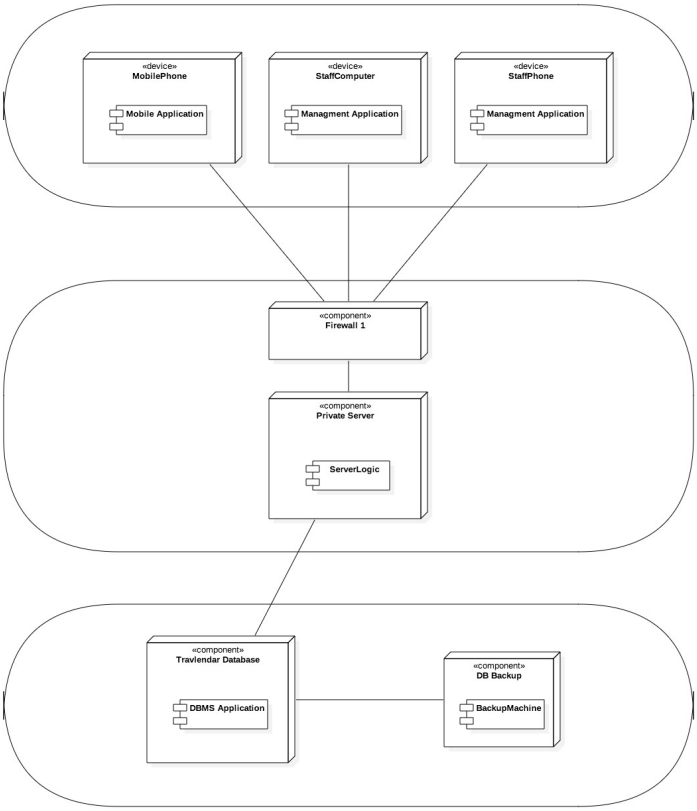


Figure 12: Deployment Diagram

2.4 Runtime view

This section shows how the components run and interact in the system, by using Sequence Diagrams.

2.4.1 Application Opening

This Sequence Diagram represents the first operations of *Travlendar+*. When application is opened, the *Auto Login* procedure in the 'Access Manager' (see figure 3) takes place: the 'Autentication Manager' checks for 'previously accesses data' in 'Saved Data Login' component. If the User has logged before, the User View is logged as explained later. On the contrary the *Login Page* of 'Guest View' of 'Mobile Application' component (see figure 2) is loaded.

On this page the User, if already registered, he must fill the login form with his credential. 'Autentication Manager' connects then to the 'DMBS' -contained in '*Travlendar* Server' (see figure 9)- in order to check if the form is properly compiled.

'DBMS' returns a Result; if the credential are correct 'Autentication Manager' notifies Guest of the success of the procedure, and saves the aforementioned data to the 'Saved Data Login' component.

Then a the *Content Recreation* procedure starts: 'Profile Manager' component - contained in 'Application Aggregator' (see figure 4)- loads all the necessary information about the User, and makes 'Calendar Manager' load all the informations about Appointments, Trips, and Breaks of the User; if necessary, 'Calendar Manager' and 'Preference Manager' (see figure 6) have to restore them from the 'DBMS'. After all the data are loaded, the component 'User View' in 'Mobile Application' is told to show the *MainPage* to the User.

In case of incorrect credentials, 'Autentication Manager' make the 'Guest View' notify the User of the not well compiled form, and makes him try again until the insertion of correct credentials.

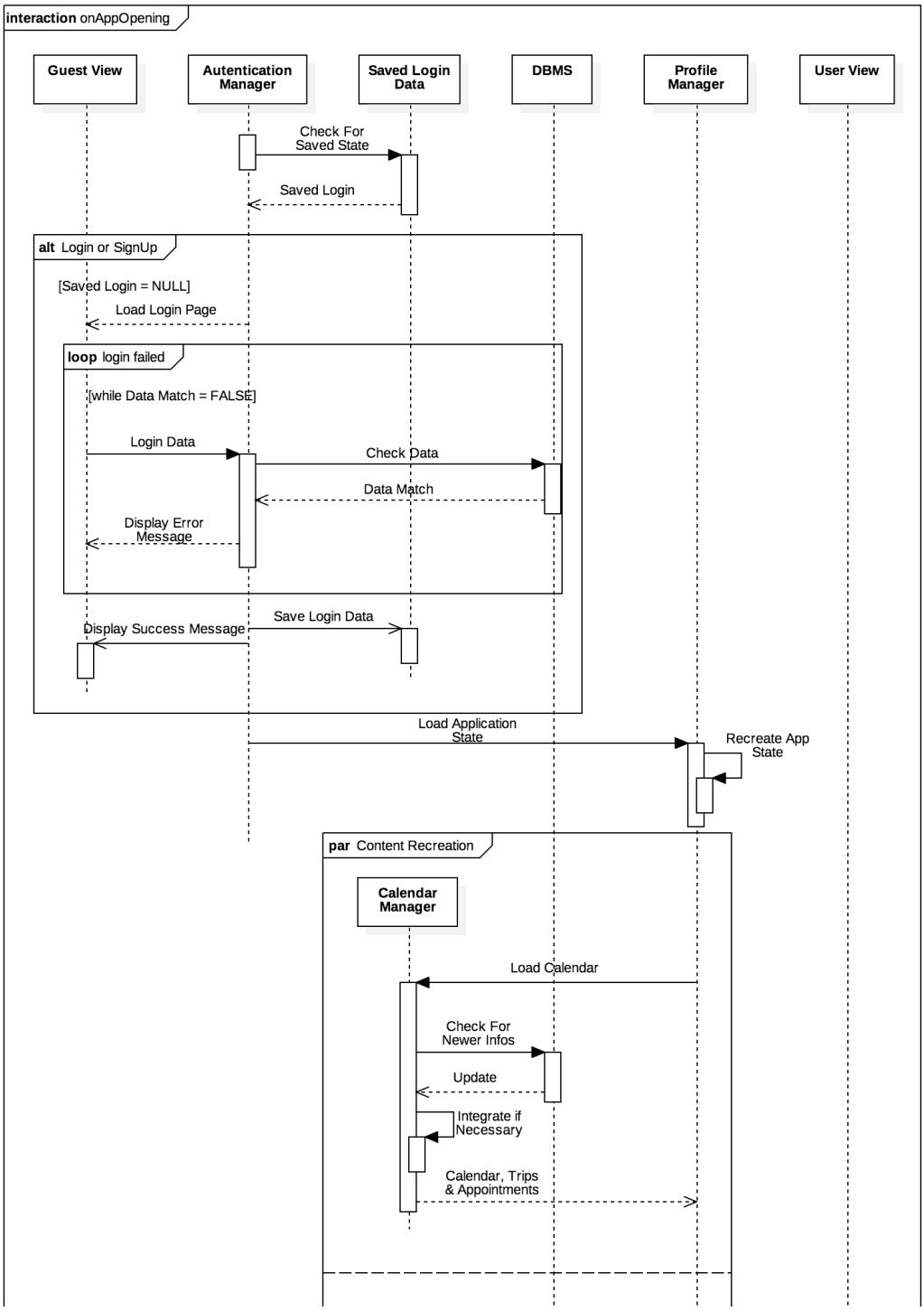


Figure 13: First Part of Application Opening Sequence Diagram

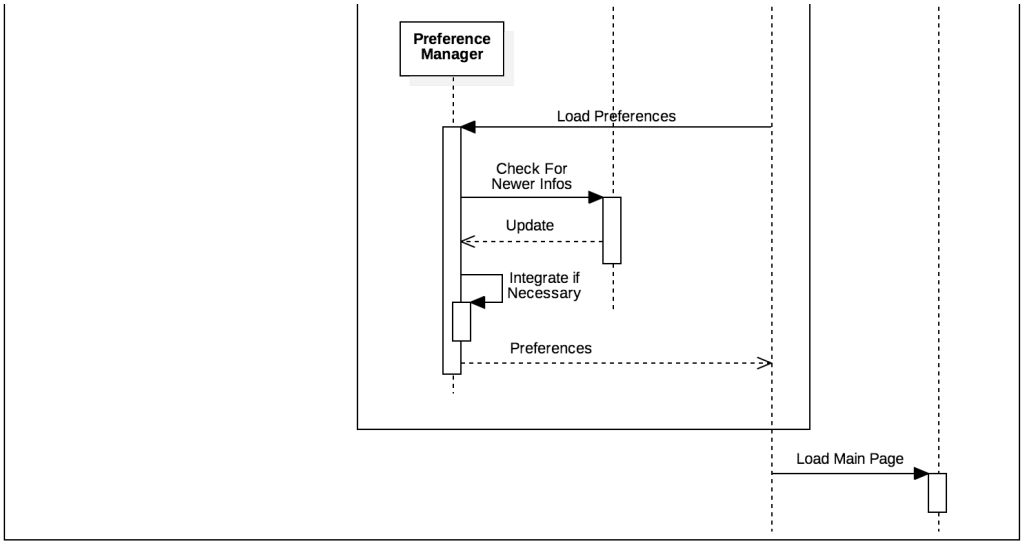


Figure 13: Second Part of Application Opening Sequence Diagram

2.4.2 Event Solutions Refreshing

This Sequence Diagram shows the relation between components when User tries to refresh or re-calculate already scheduled solutions for an Appointment. 'User Action Handler' component -contained in 'Application Aggregator' (see figure 4)- receives the User's comand, and tells the 'Appoinemnt Aggregator' component -contained in 'Calendar Manager' component (see figure 5)- to start the *Refreshing* procedure: All the Trip scheduled for the Appointment are taken from the 'Trip List' component, and are passed to the 'Scheduler' component -contained in 'Travel Logic' component (see figure 7)- in order to be confirmed o rescheduled.

'Scheduler' has to consult Excluded Vehicles, Maximum Time per Vehicle, Carbon Footprint and any other possibile preference defined by the User.

On the same time, 'Scheduler' contacts the 'API Request Distpatcher' of the '*Travlendar* Server' component (see figure 9) in order to get all the external information about the Trip, like Public Transportation Timetable, Sharing Mean Position, Weather, and any other useful detail. 'API Request Distpatcher' sends a request to 'API Manager' (see figure 10) that, through his 'Listener' contacts all the necessary external APIs, and returns the 'Scheduler' requested information. All the Trip details are passed to the 'Trip Handler' that is encharged of creating the new trip; the possible sequence of actions is shown in the RASD.

If Purchasing is required 'Trip Handler' calls the 'Payment Handler' component -contained in 'Payment Manager' (see figure 8)-; this component is encharged to open the external Application and passing all the necessary informations in order to make User purchase requested service. After procedure is completed, the external Application returns to 'Payment Handler' that store the result in 'Purchase History' component, whatever the outcome is. Indeed that if the Purchase Procedure fails, it is assumed that it will be re-tried in a second time, or a new Transportation Mean will be found, with the same sequence of actions shown in the RASD.

The just created Trip is then passed to 'Calendar Manager', that notifies the User of the calendar updates via 'User Action Handler' and 'User View', and contacts the 'DMBS' of '*Travlendar* Server' in order to store the new Trip.

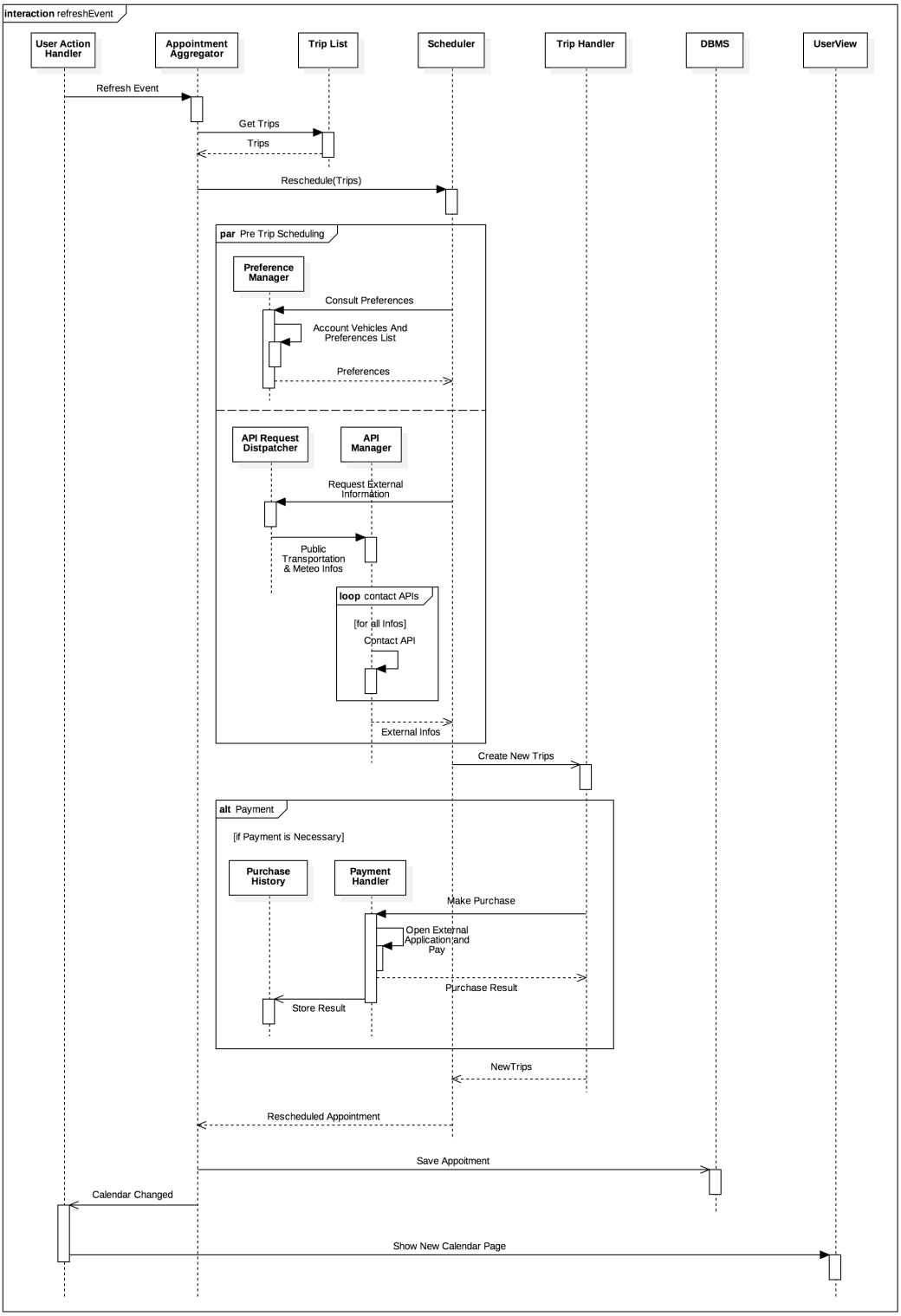
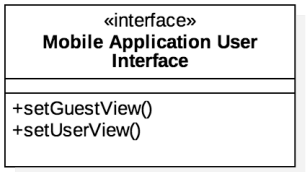


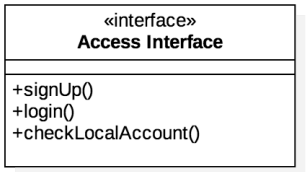
Figure 14: Event Refreshing Sequence Diagram

2.5 Component Interfaces

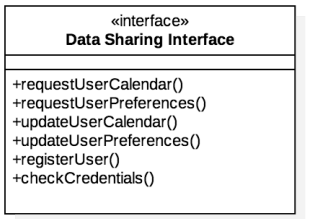
In this section we will show the component Interfaces. For each one are reported the main functionalities. Nevertheless we need to keep in mind that in further implementations these functions can be splitted into less complex ones.



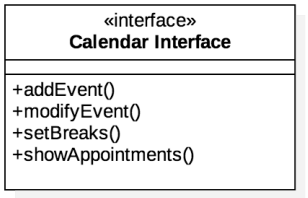
(a) Mobile Application User Interface



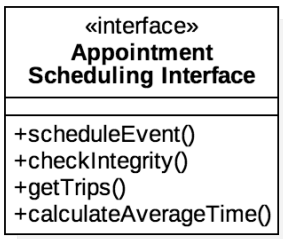
(b) Access Interface



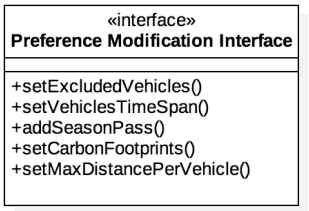
(c) Daba Sharing Interface



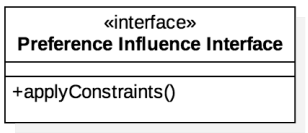
(d) Calendar Interface



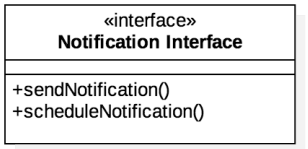
(e) Appointment Scheduling Interface



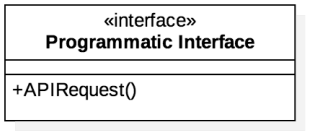
(f) Preference Modification Interface



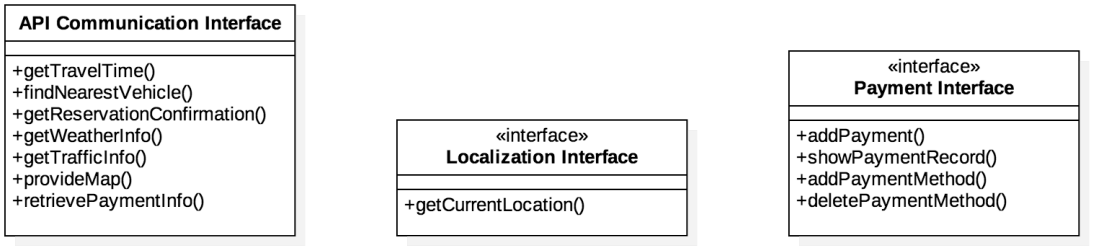
(g) Preference Influence Interface



(h) Notification Interface



(i) Programmatic Interface



(j) API Communication Interface (k) Localization Interface (l) Payment Interface

Figure 15: List of Interfaces

2.6 Architectural Styles

For our system architecture we adopt a 3-tier client/server architecture : we'll have a Mobile Application Clients (the presentation layer) and Travlendar Server (which accounts with its separate functions both for Logic layer and Data Layer).

Mobile Application Clients: This layer is represented by the *Travlendar+* Application, a thick client and an interactive and dinamic GUI. The application will be written in Java for increased portability and will be able both to geo-localize its user and to communicate with the core of the travel Logic: this is the foundation of information synching and requests forwarding to APIs.

***Travlendar+* Server:** The business Server runs the business logic. Its main duty is to forward requests of the users to the Data layer and external APIs and to send them back bound by the necessary users' constraints. These operations are executed under a microservice architectural pattern.

The Data Layer: This layer comprehends both the DBMS -which stores and manages users' preferences and timetables- and the external service agents (objects which call external services). In the former case it is required the capability of providing data through SQL queries using ODBC protocol, while in the latter it is required a good implementation of the ad-hoc functions of the external APIs.

The mobile application will be developed in Java because of the cross-compatibility and portability of the language. The Travlendar Server will be developed in JEE and will link single users with the database. The DBMS will be managed through MySQL relational language.

Patterns

Pattern are mostly required to smooth the construction of our system. In particular, we'll be using a MVC (Model - View - Control) Pattern for our mobile application, Client/Server and Single Instance component.

MVC is used to define our Java mobile application: the view will be the dynamic GUI, the Travel Logic the control and the model will be the data received from the APIs and the locally stored timetables and trips.

Single Instance component will be mostly useful when treating the components of the mobile application: components like Access Manager or Travel Logic will be the ones mainly touched by this pattern.

Client/Server pattern is the one we adopted to shape the interactions between our mobile application (the client) and the Travlendar+ Server (i.e, the server). Reasons for adoption have mainly been the simplicity and widespread use of the model.

3 Algorithm Design

In this section we will describe the most interesting algorithms, from a computational point of view; we have decided not to include algorithms for the more mechanical and common part of the system (such as login operations or interactions with the DBMS). In addition, some sequence of actions have already been described in a more general way with some Sequence Diagrams in previous parts of this document, and in the *Use Cases* section of the RASD.

3.1 Ranking

This algorithm shows how System ranks all the feasible solution based provided by the external sources, like Public Transportation, via Car, Bike or Foot, depending on:

- time needed for completing the trip.
- number of subtrips (only for Public Tranportation).
- if the Car have to be used in other trips on that day.
- money needed for the Tranportation Mean, calculated as:
  - public ticket cost for Public Services.
  - average fuel cost for Car.
  - tariff €/time for Sharing services.
- User preferences.
- Weather forecast for the Appointment day (if and only if the Appointment is no longer than 15 days).

Since solutions have been already calculated individually, is supposed that every element in the input list has been already divided in subtrips either by the External API Manager or by Scheduler (see figure 7 and figure 10) in a previous Step. Is also assumed that 'Public Service Manager' (see RASD Class Diagram) provides a list of different tranportation means, and all the consistent possible combinations of them. Every element in the input list is filtered by the 'Excluded Vehicles List' (see figure 6) For remaining elements, every inner SubTrip time is compared with the average reference time that the User have to spend by going with owned Car or bike, or, if is a short distance, by going with foot, in order to categorize solutions in *suitables*, *valid alternatives*, and *unconvenient*.

Then for every category solutions:

- items are orderd by *time needed* and *number of SubTrip*
- if User has a Season Pass of a public transportation company, relative solutions are put on the highest rank.
- solution cost are calculated, as aforementioned.

If a solutions from lower category are advantageous with respect to 'higher' solutions, those are put again in the upper category. A solution is advantageous if *time needed*s difference between the two category, defined as
$\Delta = timeNeeded(bestUpperSolution) - timeNeeded(lowerSolution)$,
is less than 15 minutes, and either is a cheaper solution or User has a Season Pass that doesn't belong to any other company of Upper-category solutions.

If the Appointment is scheduled in less than 15 days, and weather forecast is predicted al *non consistent for outdoor tripping*, solutions that expect a $total\_Outdoor\_Time$ defined as the sum of the time spent by walking and biking, greater than 3 minutes are downgraded in the respective category.

If in the other Appointments of that day a proprietary Car solution is already scheduled, 'Only by Using Car' solution will be encouraged, but if and only if:

In the end all the category *suitables* and *valid alternatives* are merged into an unique list made so that User can choose one. The algorithm return also the *bestSolution* got by 'Poping' of the first element of the list, and the solution obtained by 'Only Walking' and 'Only by Using Car'.

Algorithm 1: Rank Solution

Data: Trip, List of Calculated Solutions, Preferences, Calendar
Result: List of Ranked Solutions, Best Calculated Solution, Only Car Time, Only Walking Time

```
// Calculate 'Only Car' and 'Only Walking' Solutions
carSolution ← Scheduler.carTime;
bikeSolution ← Scheduler.walkingTime;

// Filtering Solutions based on Preferences
forall Trip in Input_List do
    if Trip.AssignedTransportationMean() is in Excluded_Vehicles_List then
        InputList.remove(Trip);
FilteredList ← InputList;

// Assign a category
forall Trip in Filtered_List do
    δ_Advantage ← 0;
    foreach SubTrip in Trip do
        switch SubTrip do
            case Long_Distance do
                AVG_Ref ←
                  arg min (Scheduler.AVG_Time(Car), Scheduler.AVG_Time(Bike));
            case Short_Distance do
                AVG_Ref = Scheduler.AVG_Time(Foot);
        comparison ← AVG_Ref − SubTrip.Time;
        δ_Advantage ← δ_Advantage + comparison;

    if δ_Advantage ≫ 0 then
        Suitable_List.add(Trip);
    else if δ_Advantage ≥ 0 then
        Valid_Alternatives_List.add(Trip);
    else if δ_Advantage < 0 then
        Unconvenient_List.add(Trip);
    else
        Filtered_List.remove(Trip);
```

```
   // Category Ranking
25 foreach Category_List do
26     Category_List.sortBy(TimeNeeded, SubTripsNumber, ascending);
27     forall Trip in Category_List do
28         cost ← calculateCost(Trip);
29         if User has Season Pass for that Trip then
30             cost ← SeasonPass.Promotion;
31 Category_List.partialOrdering(cost) ;

   // Searching for Advantageous Lower Solutions
32 bestSolution ← Suitable_List.takeFirst();
33 foreach lowerTrip in Lower_Categories_List do
34     Δ ← timeNeeded(bestSolution) − timeNeeded(LowerTrip);
35     if Δ ≤ 15 minutes then
36         if (isCheaper(bestSolution) or ∃SeasonPass(Trip) and
            ∄SeasonPass(UpperCategoryTrip)) then
37             categoryPromotion(Trip);
38 Solution_List ← join(Suitable_List, Valid_Alternatives_List);
39 Solution_List.add(carSolution, bikeSolution);

   // Car is used during the Day
40 if Calendar contains Trips with Car then
41     Solution_List.rankUp(carSolution);

   // Weather Accounting
42 if WeatherForecaster.Time() ≤ 15 days and WeatherForecaster.Prediction() is
    bad_Weather then
43     Solution_List.remove(bikeSolution);
44     forall Solution in Solution_List do
45         outdoorTime ← 0;
46         foreach SubTrip in Solution that is 'outdoor' do
47             outdoorTime ← outdoorTime + SubTrip.Time;
48         if outdoorTime ≥ 3 minutes then
49             Solution_List.rankDown(Solution);

50 return Solution_List;
```

3.2 Ensure Break Time

This Algorithm show how to ensure that, when a new Appointment is inserted, that it all the Specified Break Time are ensured (See RASD for more Details). In order to apply this algorithm, the Appointment is supposed to be inserted during the *Time Span* of a Break

Given the actual Calendar, including Appointment, Trips and Breaks of a particular Day (see figure 5), the insertion of a new Appointment is **supposed not to be overlapped with other Appointments**.

With respect to the new Appointment,

- If a previous Appointment exists, the estimated time to reach the Location from the event is added before the $newAppointment.StartHour$.
- If the next Appointment is right after the 'End Hour', the estimated time to reach the new Location is added after the $newAppointment.EndHour$.
- In the Other case this time is added before the 'Start Hour' of the next Event.

Then, the new Appointment is added, to the Timeline of the Day and a new Break is calculated in order to be consistent:

- if the new Appoiment $StartHour$ and $EndHour$ are far from the $Break\_Start$ and $Break\_End$ the Break is left alone and the Algorithm returns previously computed Break.
- A time interval δt is determined, where the left extremity is the $Time\_Span\_Init$ or the $Appointment.EndTime$, while the right extremity is the $newAppointment.StartTripTime$. If δt is greater or equal than Break's $Minimum\_Duration$, is added to a $Possible\_Break\_List$.
- whether a possible Break is found or not, the process is reiterated with a new δt where where the left extremity is the $newAppointment.StartTripTime$ or the $nextAppointment.EndTime$, until the right extremity is the $lastAppointment.StartTripTime$ or the $Time\_Span\_End$

At the end, the maximum interval is took from $Possible\_Break\_List$ and it's set as $Default\_Break$ If $Possible\_Break\_List$ is empty the User is Notified that the new Appointment can't add that appointment because it doesn't let him have a Break.

Algorithm 2: Ensure Break Time

Data: Daily Calendar, New Appointmend
Result: Daily Calendar consistent with breaks

```
// Adding Extra Travel Time to New Appointment

// Previous Appointmen before New Appointment
if Calendar contains an Appointment Event such that
 Event.EndTime − NewAppointment.StartTime ≤ 1hour then
    timeSpan ←
     Scheduler.AVGTime(Event.Location, NewAppointment.Location);
    NewAppointment.StartTime ← NewAppointment.StartTime − timeSpan;
// Next Appointmen is right after New Appointment
if Calendar contains an Appointment Event after NewAppointment then
    timeSpan ←
     Scheduler.AVGTime(NewAppointment.Location, Event.Location);
    if NewAppointment.EndTime − Event.StartTime ≤ 1hour then
        NewAppointment.EndTime ← NewAppointment.EndTime + timeSpan;
    else
        Event.StartTime ← Event.Start − timeSpan;

// Check Break Consistency
forall Breaks do
    if Appointment's StartTime or EndTime are near Break's StartTime or
     EndTime then
        okFlag ← False;
if okFlag then
    if Added timeSpan then
        removeTimeSpans();
    Calendar.addAppointment(newAppointment);
    return Calendar;

// Check for New Breaks
forall EndTime, StartTime in Break.TimeSpan do
    δt = EndTime − StartTime;
    if δt ≥ Break.miminumDuration then
        Possible_Break_List.add(EndTime, StartTime, δt);

if Possible_Break_List.isNotEmpty() then
    return arg max(Possible_Break_List);
else
    NotificationManager.notify (Appointment doesn't permit Breaks);
```

4 User Interface Design

This section is a reference to section 3.1.1 "User Interfaces" already seen in the RASD. It was decided not to introduce new interfaces nor mockups as the existing ones are quite exhaustive with regards to the goals. In particular, it was already shown how interfaces were going to look like : we refer to mockups dealing with "login", "create an appointment", "weekly view" and all possible solutions to reach an event. Notice that these are the goals the essential application functions from user's perspective

5 Requirement Traceability

The Design Document at hand was thought and developed in the scope of fulfilling optimally the requirements specified in its corresponding RASD. While the RASD gives a comprehensive list of the various Goals and requirements, here we'll only display how the mapping between the two documents was done, showing the bare minimum information concerning the latter document.

*[G1]*System provides an authentication system.

- R.1.1 System provides sign-up and an authentication mechanism. **This requirement is mapped into Access Manager component**.
- R.1.2 System requires a unique username and a password for every user. **This requirement is mapped into Travlendar Server component**.
- R.1.3 An unregistered user is locked out the application and can only see the registration page. **This requirement is mapped into Mobile Application and Access Manager components**.
- R.1.4 User has to confirm by mail his registration. **This requirement is mapped into Travlendar Server component**.
- R.1.5 Only a correct combination of username and password will grant access. **This requirement is mapped into an Access Manager and Travlendar Server component**.
- R.1.6 Application will implement a password retrieval mechanism. **This requirement is mapped into Travlendar Server component**.
- R.1.7 Each modification made to a user account must be saved into Travlendar+ Server to be made effective. **This requirement is mapped into Travlendar Server and Access Manager component**.
- R.1.8 New user registration is successful only after data is stored on Travlendar+ Server and a confirmation is received by the system. **This requirement is mapped into Travlendar Server component**.

[G2] The application integrates a time-slot based system for appointments.

- R.2.1 The calendar integrates a calendar and a timetable.

R.2.2 Calendar must give to the user granularity regarding both months and days. **These requirements are mapped into the Calendar Manager components**.

R.2.3 Calendar and Timetable can be modified only by the user inserting events. No one else is allowed to either see or modify the information they contain. **This requirement is mapped into Calendar Manager and Application Aggregator components**.

R.2.4 Calendar and Timetable for each user are remotely copied on Travlendar+ Server every time a user creates/modifies/deletes an event.

This requirement is mapped into Calendar Manager and Travlendar Server components.

[G3] Registered User can create appointments.

R.3.1 User has to be registered and logged in the system in order to create an appointment. **This requirement is mapped into Calendar Manager and Application Aggregator components**.

R.3.2 Appointments can be divided into work appointments (or meetings) and personal appointments.

R.3.3 Appointments require a location, a starting time and an end time. **These requirements are mapped into the Calendar Manager component**.

R.3.4 Appointments location must be within the boundaries of the operative zone. **This requirement is mapped into Calendar Manager and Travel Logic components**.

R.3.5 There cannot be appointments with the same name, location and time. **This requirement is mapped into Calendar Manager component**.

R.3.6 System must check suitability of new entries based on already existing appointments. **This requirement is mapped into Calendar Manager and Travel Logic components**.

R.3.7 Appointment start time can't precede the actual system time at the moment of inserting it.

R.3.8 User can select favourite travel means and priority for each appointment.

R.3.9 Each appointment must be associated to a level priority . **These requirements are mapped into Calendar Manager component**.

R.3.10 The creation of an appointment must be remotely saved on Travendlar+ server in order to be successful and complete. **This requirement is mapped into Calendar Manager and Travlendar Server components**.

[G4] Registered Users can edit appointments.

R.4.1 A modified meeting must respect all the constraints imposed during the creation of a new meeting, as the requirements in *[G3]*.

R.4.2 A meeting can be modified up until its end time.

R.4.3 If the meeting is modified, the system behaves as if such an event was inserted for the first time, calculating all possibile conflicts with pre-existing events.

R.4.4 No limit actually exists on the amount of times an event can be modified within the aforementioned constraints. **These requirements are mapped into Calendar Manager and Application Aggregator components**.

R.4.5 A modification must be correctly saved on the remote *Travlendar+* server in order to be succesful and completed. **This requirement is mapped into Calendar Manager and Travlendar Server components**.

R.4.6 Deleting an appointments must belong to the set of modifications. **This requirement is mapped into Calendar Manager component**.

[G5] The application can automatically compute a personalized selection of travel times between appointments to choose from.

R.5.1 The application must refer to Travel Logic for the expected travel time.

R.5.2 The application must be able to suggest a combination of various means to reach the desired destination.

R.5.3 In case the trip expects more than one travel mean, the journey must be divided into sub-problems whose expected travel time has to be calculated. Same goes with public means stop and shared vehicles. **These requirements are mapped into Travel Logic component**.

R.5.4 Starting location for travel can be inserted manually, retrieved by the previous event or calculated through geo-localization. **This requirement is mapped into Travel Logic and Localization Manager components**.

R.5.5 The application must rank the suggestions according to their priority, presence of preferred travel means and time required. **These requirements are mapped into Travel Logic component**.

R.5.6 The registered user must be able to choose to filter out specific travel means.

R.5.7 Favourite travel means associated to an appointment must always show up. **These requirements are mapped into Travel Logic and Preference Manager components**.

R.5.8 In case two or more appointments overlap, an appointment with higher priority is considered automatically chosen and all the remaining ones are arranged according to their priority. Warnings must follow as expected. **This requirement is mapped into Travel Logic and Notification Manager components**.

R.5.9 The route can include intermediate destinations before the final, target one.

R.5.10 When a shared vehicle is suggested the parking zone nearest to the destination must be always inserted among the intermediate destinations.

These requirements are mapped into Travel Logic component.

R.5.11 The sytem must grant to know daily scheduled times for public transportation through its APIs. **This requirement is mapped into API Manager component**.

R.5.12 When the starting time of a trip associated to an event is only one hour away the system must notify the user with an updated list of travel time so he can choose. **This requirement is mapped into Travel Logic and Notification Manager components**.

R.5.13 According to real world data, each travel must have associated to itself the carbon footprints. **This requirement is mapped into Travel Logic and Preference Manager components**.

R.5.14 Travels that do not satisfy all User's contraints must be excluded. **This requirement is mapped into Travel Logic and Preference Manager components**.

[G6] User can choose a solution among the scheduled ones.

R.6.1 Selecting a solution that is not a personal vehicle must show both intermediate and final destinations. **This requirement is mapped into Calendar Manager and Travel Logic components**.

R.6.2 The application must arrange a navigable interface of feasible solutions. **This requirement is mapped into Mobile Application, Application Aggregator and Calendar Manager components**.

R.6.3 Choosing a solution that includes a public transportation mean must show the user the possibility to buy a ticket. In case of ticket purchase *Travlendar+* checks if the mobile app corresponding to the desired services is installed on the system. All the following steps take place within such an environment, until control is returned to *Travlendar+*. **This requirement is mapped into Travel Logic components**.

R.6.4 Choosing a solution that includes a shared vehicle must show the user the possibility to locate and rent such a vehicle. **This requirement is mapped into Travel Logic and API Manager components**.

R.6.5 Choosing a solution must not be definitive. **This requirement is mapped into Calendar Manager and Travel Logic components**.

R.6.6 System must recognize by itself through geolocalization that a user reached destination; also, User must always be able to stop the trip. **This requirement is mapped into Localization Manager and Travel Logic components**.

[G7] The application warns the user if locations are unreachable in the allotted time.

R.7.1 The application must realize if the alloted time is sufficient from either the last event, current location or manually inserted location. **This requirement is mapped into Travel Logic and Localization Manager components**.

R.7.2 The application must use as a reference the time to cover distance between the starting place and the destination one, using the futured scheduled time for public transportation if necessary. **This requirement is mapped into Travel Logic and API Manager components**.

R.7.3 Warning must arrive also while on the road if the travel mean is no longer suitable, or the best solution: in that case the system is going to prompt a new eventual choice of travel means. **This requirement is mapped into Notification Manager, Travel Logic, and Localization Manager components**.

R.7.4 When user reaches destination warnings must stop automatically. **This requirement is mapped into Notification Manager and Localization Manager components**.

R.7.5 Warnings can be disabled on the road by the user. **This requirement is mapped into Application Aggregator component**.

[G8] Allow users to put constraints on different travel means and limit carbon footprints.

R.8.1 User must be able to rule out vehicles from search result returned by the system scheduler.

R.8.2 When the option of limiting carbon footprints gets enabled the associated CO2 consumed by each travel must be taken into account in travels scheduling.

R.8.3 User must be able to put a constraint on the number of travel means adopted for a single travel.

R.8.4 User must allow at least a single travel mean.

R.9.5 User cannot remove "walking" from travel mean preferences. **All these requirements are mapped into Travel Logic and Preference Manager components**.

[G9] The application features additional User's breaks.

R.9.1 Each Break is characterized by a duration, the time of the day they start in and by the time frame within are allowed.

R.9.2 Breaks can be periodic.

R.9.3 System reserves a minimum quantity of time which is not shorter than the break duration.

R.9.4 Breaks must be completely encapsulated within the time frames the break is allowed in. **These requirements are mapped into Calendar Manager component**.

R.9.5 Within the time frame of a break the scheduler must always grant a free time span whose duration must be at least equal to the corresponding break duration. **This requirement is mapped into Calendar Manager and Travel Logic components**.

[G10] The application allows to buy tickets for public services.

R.10.1 Buying a ticket must reroute the user to the corresponding mobile application, after system has succesfully checked it is installed.

This requirement is mapped into Payment Manager component.

R.10.2 Purchase confirmation and additional data about the payment is retrieved through API requests.

This requirement is mapped into API Manager and Payment Manager components.

[G11] The application allows the nearest shared vehicle to be found and reserved.

R.11.1 A shared vehicle must necessarily belong to a bike-sharing service or a car-sharing service.

R.11.2 All services linked to shared vehicles must be automatically disabled if the location of an appointment is out of the boundaries of the influence zone.

These requirements are mapped into API Manager and Travel Logic components.

R.11.3 All sharing services have their own API which is used by the system to locate the vehicles and retrieve confirmation of reservation.

This requirement is mapped into API Manager component.

R.11.4 The external service can communicate with our mobile application. In case of reservation *Travlendar+* checks if the mobile app corresponding to the desired services is installed on the system. All the following steps take place within such an environment, until control is returned to *Travlendar+*. **This requirement is mapped into Travel Logic component**.

R.11.5 The location of all the vehicles must be shown in the same interface, merging data from different APIs.

R.11.6 Only shared vehicles that are free and available must be displayed and possibly reserved. **These requirements are mapped into API Manager component**.

[G12] The application allows the user to oversee his position in real-time as well as the route of his travel.

R.12.1 Application integrates a map system submitted by GMAPS API.

R.12.2 User must be able search for a specific location. **These requirements are mapped into API Manager and Travel Logic components**.

R.12.3 The mobile device must be able to track its current position through geo-localization. **These requirements are mapped into API Manager and Localization Manager components**.

R.12.4 Positions out of the operative zone can't be accepted by the system and won't be displayed. **This requirement is mapped into Travel Logic component**.

[G13] The User can submit additional preferences.

R.13.1 User must be able to forbid travel means within time spans, also periodical ones.

R.13.2 User must be able to put a constraint on the maximum amount of space and time he can give to each travel mean.

R.13.3 User must be able to link one or more season passes to his account. **These requiremente are mapped into Preference Manager component**.

R.13.4 User must be able to link one or more credit cards to his account. **This requirement is mapped into Payment Manager component**.

R.13.5 Each modification apported by the User to its additional preferences is only made effective when synced on *Travlendar+* Server. **This requirement is mapped into Preference Manager and Travlendar Server components**.

6 Implementation, Integration and Test Plan

6.1 Implementation Plan

- Listener (API Manager)
- API Request Dispatcher (Travlendar Server)
- OpenWeatherMap API (API Manager)
- Google Maps API (API Manager)
- Google Transit API (API Manager)
- CAR2GO API (API Manager)
- Other API-Based system (API Manager)
- DBMS (Travlendar Server)
- Autentication Manager (Access Manager)
- Profile Manager (Application Aggregator)
- User Action Handler (Application Aggregator)
- SignUp Handler (Access Manager)
- Saved Login Data (Access Manager)
- Appointment Aggregator (Calendar Manager)
- Preference Handler (Preference Manager)
- Season Pass Handler (Preference Manager)
- Excluded Vehicles List (Preference Manager)
- Preference List (Preference Manager)
- Trip List (Calendar Manager)
- Break List (Calendar Manager)
- Scheduler (Travel Logic)
- Trip Handler (Travel Logic)
- Payment Handler (Payment Manager)
- Credit Card List (Payment Manager)
- Notification Manager
- Localization Manager

- Guest View (Mobile Application)
- User View (Mobile Application)

6.2 Integration Strategy

6.2.1 Entry Criteria

In order to start a correct integration test we must ensure the following conditions are granted:

- Requirements Analysis and Specification Document as well as the Design Document must be complete in each section and up to date, reflecting the current state of the project
- All sub-components must be already unit tested and bug free
- Risk Assessment has been layed out since it'll serve us in critical-first module integration

In addition to this we assume that all algorithms that interface external agents and make use of APIs will be already tested and that they will work as intended in receiving external data when proceeding with integration (we're talking about all sub-components that contain the word 'API' in API Manager component).

6.2.2 Elements to be integrated

All components we created and delimited in components diagram are going to be integrated. All components that give the chance to do so, will also be tested regarding the integration of the single subcomponents.

6.2.3 Integration Testing Strategy

The integration of the system will be guided by a bottom-up approach. This strictly comes from an analysis of the dependency structure of the components diagram provided in the DD, restructured in Figure 16 and Figure 17 to better highlight the key modules and their connections.

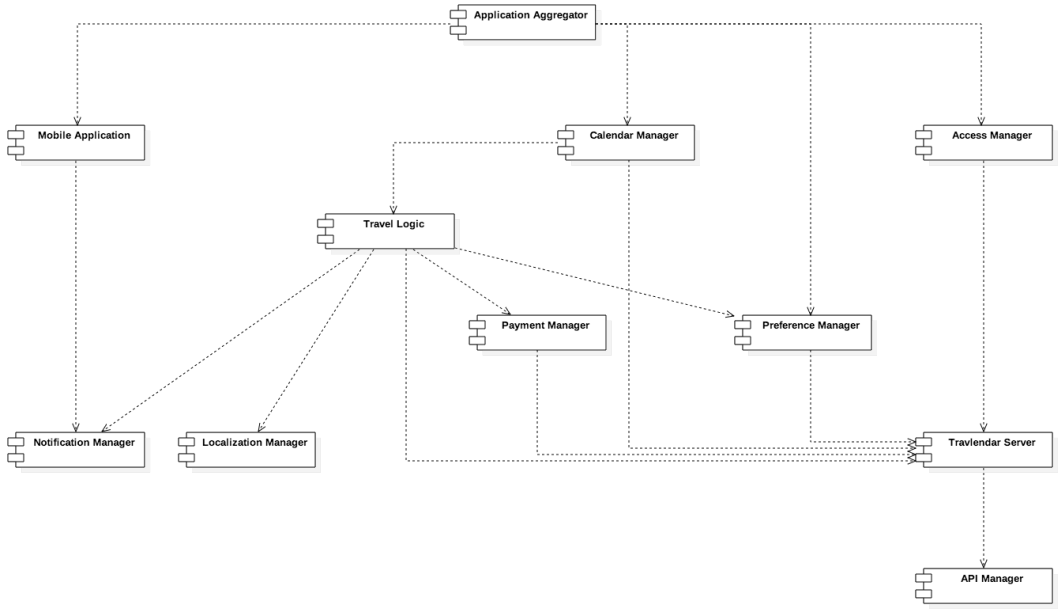


Figure 16: Dependency tree of components: high level view.

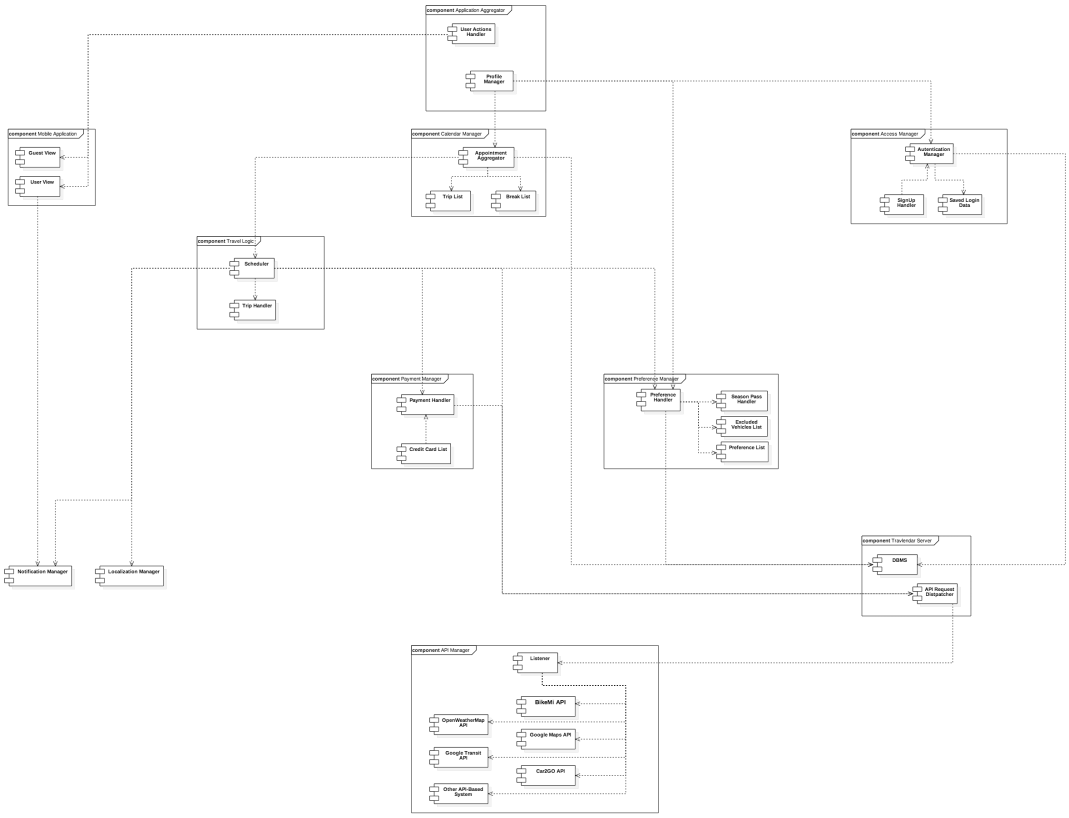


Figure 17: Dependency tree of components: complete system view.

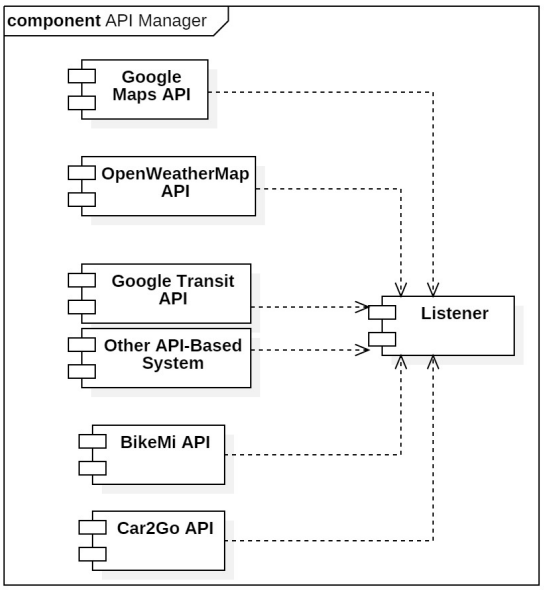
6.2.4 Sequence of component integration

From a component perspective we can safely say we adopted a bottom-up policy for the integration plan, while we also used a critical-first approach in sub-components integration (requiring, therefore, stubs in such cases). Our choice has mainly been driven by its ease and by the relatively small scale of our system.

Sequence of component/function integration. It's time for us to detail the integration order of our components, delving in what we simply anticipated with our dependency diagrams.

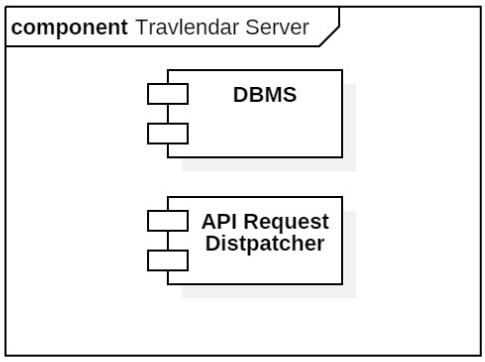
Component : API Manager.
Internal integration strategy: Bottom - Up.
Integration order:

- Google Maps API.
- OpenWeatherMap.
- Google Transit API.
- Car2Go API.
- BikeMi API.
- Other API-Based System.
- Listener.



Component : Travlendar Server.
Internal integration strategy: Critical - First.
Integration order:

- DBMS.
- API Request Dispatcher.

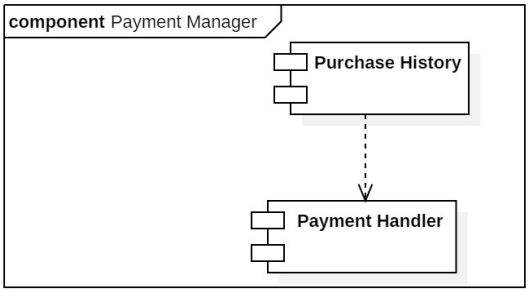


Component : Localization Manager doesn't need sub-components integration plan.

Component : Notification Manager doesn't need sub-components integration plan.

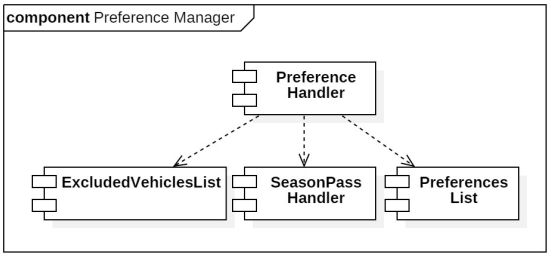
Component : Payment Manager.
Internal integration strategy: Bottom - Up.
Integration order:

- Payment Handler.
- Purchase History.



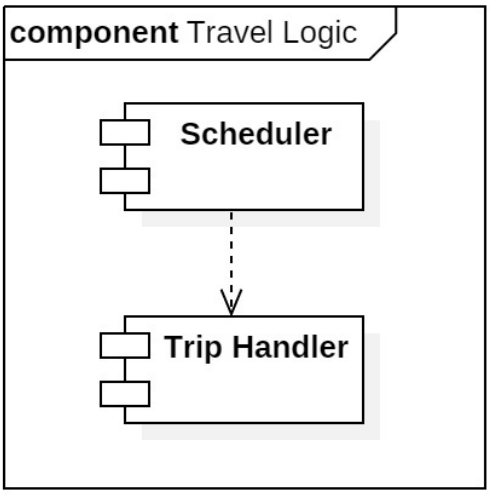
Component : Preference Manager.
Internal integration strategy: Bottom - Up
Integration order:

- Excluded Vehicles List.
- Season Pass Handler.
- Preferences List.
- Preference Handler.



Component : Travel Logic.
Internal integration strategy: Bottom - Up.
Integration order:

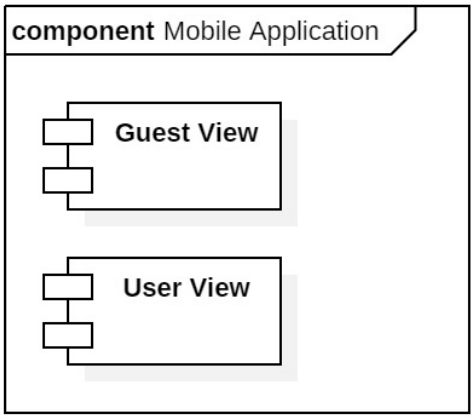
- Trip Handler.
- Scheduler.



Component : Mobile Application.

Internal integration strategy: Critical - First.
Integration order:

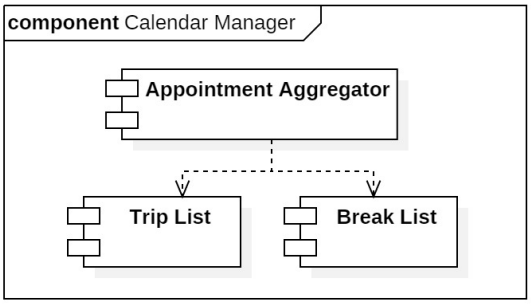
- User View.
- Guest View.



Component : Calendar Manager.

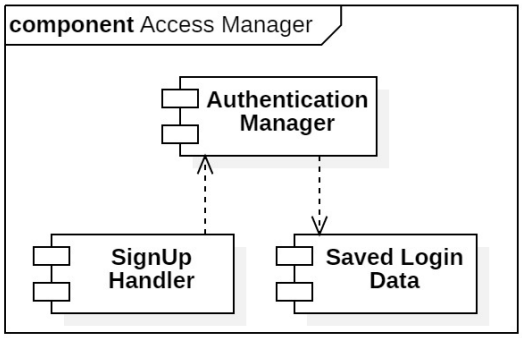
Internal integration strategy: Bottom - Up.
Integration order:

- Trip List.
- Break List.
- Appointment Aggregator.



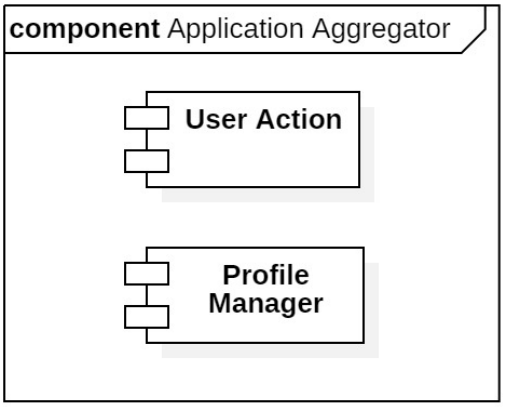
Component : Access Manager.
Internal integration strategy: Critical - first.
Integration order:

- Authentication Manager.
- SignUp Handler.
- Saved Login Data.



Component : Application Aggregator.
Internal integration strategy: Critical-First.
Integration order:

- Profile Manager.
- User Actions Handler.



6.2.5 Subsystem integration sequence

The integration of the macro-components is performed, as said, in a bottom-up fashion. Below are the 6 steps in which this process has been divided for scheduling reasons. Inside a single step, multiple macro-components participate in the integration, each one relying on the macro-components in the previous levels.

Step 1

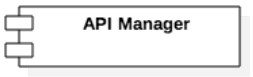


Figure 18: Subsystem integration sequence: step 1.

Step 2

Figure 19: Subsystem integration sequence: step 2.

Step 3

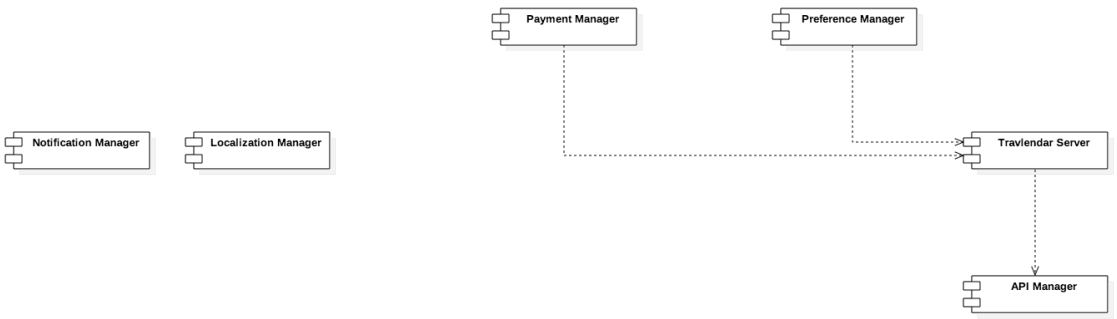


Figure 20: Subsystem integration sequence: step 3.

Step 4

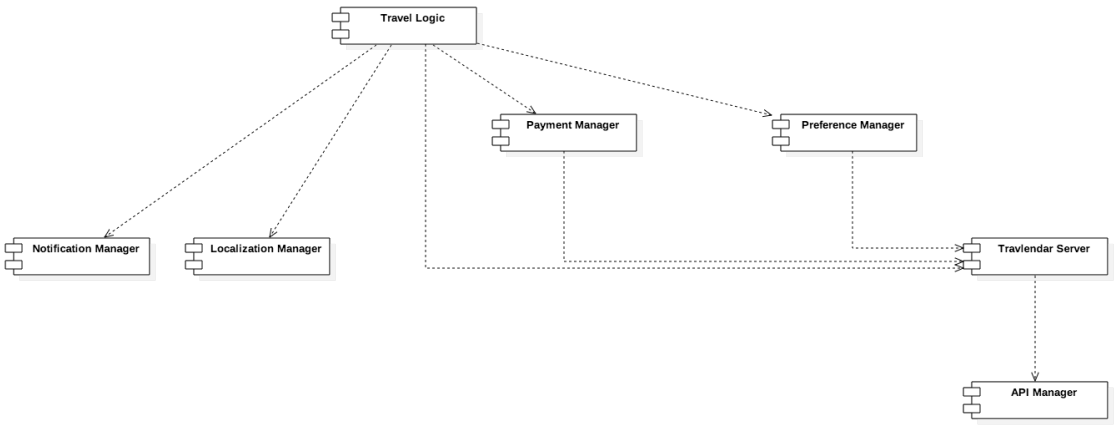


Figure 21: Subsystem integration sequence: step 4.

Step 5

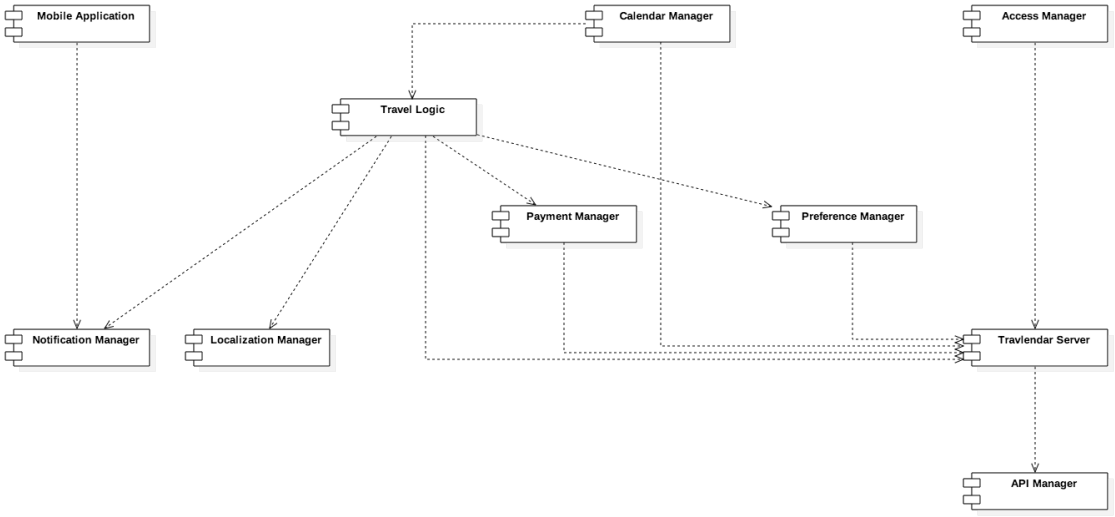


Figure 22: Subsystem integration sequence: step 5.

Step 6

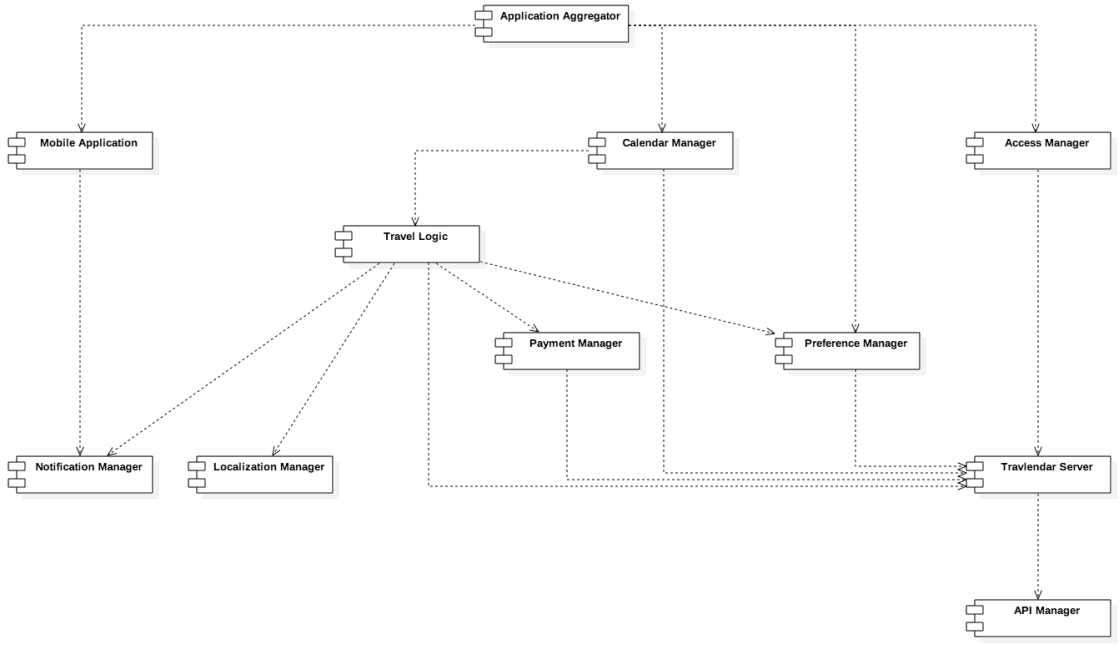


Figure 23: Subsystem integration sequence: step 6.

6.3 Individual Steps and Test Description

The components will be tested following the interfaces they share and that we previously described, testing core methods also in case of sub-components integration. Our main goal is to spot any kind of faults and failures, focusing in particular on input domains, covering the restricted spectrum of system reactions. Because of this, we'll detail the tests in easy-to-consult tables, each introduced by the components and subcomponents that are required. Each table represents an interface method, and on two columns we'll detail each input and its expected result. We'll follow a bottom-up approach that requires the creation of drivers from time to time.

6.3.1 API Communication Service

OpenWeatherMap API,Listener

getWeatherInfo()	
Input	**Effect**.
NullArgument.	NullArgumentException is raised.
Invalid Location .	OpenWeather services return the information cannot be retrieved and the Listener signals the location is wrong.
Valid Location.	Listener retrieves the required data structure from OpenWeatherMap and formats it.

GoogleMapsAPI, Listener

getTravelTime()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
Invalid Start Location/Destination Location.	GMAPS APIs signals one (or both) the locations are wrong and the Listener echoes this back.
Valid Start Location and Destination Location.	Listener retrieves the required data structure from GMAPS API.

Car2Go API, Listener
BikeMi, Listener
Other API-Based Systems, Listener

findNearestVehicle()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
Invalid Location or Location out of the boundaries.	The external signals the location is out of its service boundaries.
Valid Location.	Listener retrieves the required data structure from the external API and formats it in order for it to be navigable within the mobile application.

Car2Go API, Listener
BikeMi, Listener
Other API-Based Systems, Listener

getReservationConfirmation()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
Invalid Reservation Code.	The external service can't provide reservation data, and it sends only a message error, that Listener echoes back.
Valid Reservation Code.	Listener retrieves the required data structure from the external API and formats it in order for it to be navigable within the mobile application.

Google Transit API, Listener

getTrafficInfo()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
Invalid Path.	GMAPS APIs can't find the route and send back a message to document the impossibility of monitoring the traffic. Listener echoes this back.
Valid Path	Google Transit API provides the required information about traffic and Listener sends them back.

Listener, API Request Dispatcher

provideMap()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
Start Location and End Location.	Path from Start Location until End Location is raised.
Location and Firm's name.	Path from Location until nearest vehicle is raised.
Location and Day.	Weather information is shown.

retrievePaymentInfo()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
Valid parameters.	A request of payment is sent.
Invalid parametert.	InvalidArgumentException is raised.

6.3.2 Travlendar Server

API Request Dispatcher, Payment Handler

APIRequest()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
Valid Argument.	A payment request is sent to the Server.
Invalid Argument.	InvalidArgumentException is raised.

API Request Dispatcher, Scheduler

APIRequest()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
Valid Combination of Location, Data and Vehicle's Firm is allowed.	A request is sent to the respective dispatcher.
Invalid Argument.	InvalidArgumentException is raised.

DBMS, Authentication Manager

requestUserCalendar()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
Invalid Argument.	InvalidArgumentException is raised.
User Credentials.	An access request is sent to DBMS.

DBMS, Autentication Manager

requestUserPreferences()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
Invalid Argument.	InvalidArgumentException is raised.
User Credentials.	An access request of user preferences is sent to DBMS.

DBMS, Autentication Manager

updateUserCalendar()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
Invalid Argument.	InvalidArgumentException is raised.
User Credentials and New DataCalendar.	User Calendar is update.

DBMS, Autentication Manager

updateUserPreferences()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
Invalid Argument.	InvalidArgumentException is raised.
User Credentials and New DataPreferences.	User Preferences is update.

DBMS, Autentication Manager

registerUser()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
Invalid Argument.	InvalidArgumentException is raised.
Valid Combination of User Credential and User Information.	New User Profile is created.

DBMS, Autentication Manager

checkCredentialUser()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
Invalid Argument.	False.
User Credentials insert	True.

6.3.3 Payment Manager

Payment Handler, Purchase History

addPaymentMethod()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
Credit Card Data and linked Transaction information.	Data is stored in Purchase History.
Invalid Data.	Nothing changes and the input is rejected.

Payment Handler, Purchase History

deletePaymentMethod()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
Credit Card Data.	Credit Card Data and the linked transactions stored in Purchase History are deleted.
Invalid Input.	Everything is unchanged in Purchase History.

Payment Handler, Purchase History

showPaymentRecord()	
Input.	**Effect**.
NullArgument.	The payment record is returned in the correct data structure.
Any Parameter.	InvalidArgumentException is raised.
Invalid Input.	InvalidArgumentException is thrown and everything is unchanged in Purchase History.

Payment Handler, Scheduler

addPayment()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
Valid Ticket.	A payment request is shown.
Invalid Input.	invalidArgumentException is thrown and everything is unchanged in Purchase History.

6.3.4 Preference Manager

Preference Handler, Excluded Vehicle List, ProfileManager

setExcludedVehicles()	
Input.	**Effect**.
NullArgument.	NullArgumentException is thrown.
Argument not corresponding to a collection of vehicle objects.	InvalidArgumentException is thrown.
Valid collection of vehicle objects.	The list of excluded vehicles is updated to become as the input.

Preference Handler, SeasonPass Handler, ProfileManager

setSeasonPass()	
Input.	**Effect**.
NullArgument.	NullArgumentException is thrown.
Argument not corresponding to a collection of seasonPasses.	InvalidArgumentException is thrown.
Valid collection of seasonPasses.	The list of seasonPasses is updated to become the same as the input.

Preference Handler, Preferences List, ProfileManager

setVehicleTimeSpan()	
Input.	**Effect**.
NullArgument.	NullArgumentException is thrown.
Argument is not a collection of pairs of vehicles and time spans.	InvalidArgumentException is thrown.
Argument is a valid collection of pairs of vehicles and time spans.	The list of pairs of vehicles and time spans is updated to become the same as the input.

setCarbonFootprints()	
Input.	**Effect**.
NullArgument.	Carbon footprints are not set.
Invalid non-integer argument.	InvalidArgumentException is thrown.
Valid integer input.	Carbon footprints are set as the input commands.

setMaxDistancePerVehicle()	
Input.	**Effect**.
NullArgument.	NullArgumentException is thrown.
Argument is not a collection of pairs of vehicles and integers.	InvalidArgumentException is raised.
Argument is a valid collection of pairs of vehicles and integers.	For each specified vehicle it is set the maximum allowed distance that it can travel.

PreferenceHandler, Scheduler

applyConstraints()	
Input.	**Effect**.
NullArgument.	NullArgumentException is thrown.
Invalid Argoument.	InvalidArgumentException is thrown.
Valide combination of User Credentials and Type of constraint.	The list of trips is filtered according to existing preferences.

6.3.5 Travel Logic

Scheduler, Appointment Aggregator

scheduleEvent()	
Input.	**Effect**.
NullArgument.	NullArgumentException is thrown.
Argument is not a valid event paired with a location and a date.	InvalidArgumentException is thrown.
Argument is a valid event paired with a location and a date.	The event is added and scheduled by the logic of the mobile application.

checkIntegrity()	
Input.	**Effect**.
NullArgument.	NullArgumentException is thrown.
Argument is not a valid event paired with a location and a date.	False.
Argument is a valid event paired with a location and a date.	True.

getTrips()	
Input.	**Effect**.
Invalid Argument.	InvalidArgumentException is thrown.
Nothing.	All information about the trip.

calculateAverageTime()	
Input.	**Effect**.
NullArgument.	NullArgumentException is thrown.
Invalid Argument.	InvalidArgumentException is thrown.
Valid combination of Data, Start Location and End Location.	Average Time of trip from Start Location to End Location.

6.3.6 Calendar Manager

Appointment Aggregator, Trip List

addEvent()	
Input.	**Effect**.
NullArgument.	NullArgumentException is thrown.
Argument is not a valid event paired with a location and a date.	InvalidArgumentException is thrown.
Argument is a valid event paired with a location and a date.	The event is added to the events list, yet they miss trips, that must be added through Scheduler and Travel Logic.

Appointment Aggregator, Trip List

modifyEvent()	
Input.	**Effect**.
NullArgument.	NullArgumentException is thrown.
Argument is not a valid field or set of fields of a valid event.	InvalidArgumentException is thrown.
Argument is a valid field or set of fields of a valid event.	The event is updated as requested.

Appointment Aggregator, Break List

setBreaks()	
Input.	**Effect**.
NullArgument.	NullArgumentException is thrown.
Argument is not a valid break object.	InvalidArgumentException is thrown.
Argument is a valid collection of break objects.	The Break lists encompasses only the breaks listed in the input.

Appointment Aggregator, User Actions Handler

showAppointments()	
Input.	**Effect**.
Invalid Argument.	InvalidArgumentException is thrown.
Nothing.	List of appointments.

6.3.7 Access Manager

Authentication Manager, Saved Login Data

checkLocalAccount()	
Input.	**Effect**.
NullArgument and no login data stored.	The standard login procedures that interleave with the Travlendar Server begin.
NullArgument and no login data stored.	The standard login procedure is bypassed and the user automatically acesses the mobile application.

Authentication Manager, SignUp Handler, Profile Manager,DBMS

signUp()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
InvalidArgument Exception.	InvalidArgumentException.
Valid combination of User Credentials and User Information.	User is signed up.

Authentication Manager, Profile Manager, DBMS

login()	
Input.	**Effect**.
NullArgument.	NullArgumentException is raised.
InvalidArgument Exception.	InvalidArgumentException.
Valid combination of User Credentials.	User is logged in.

7 Appendix

List of Figures

List of Algorithms

7.1 Used tools

For this assignment, we used the following tools:

LaTeX The group used LaTeX to structure the final document and to help with versioning.

Github We leaned on Github for versioning and coordinating synchronized work.

StarUML We used StarUML to make Use Case, Class and Sequence Diagrams. StarUML.

7.2 Hours of work

Bisica, Leonardo around 44 hours of work;

Castellani, Alessandro around 46 hours of work;

Cataldo, Michele around 42 hours of work.