

Politecnico di Milano

A.Y. 2017/2018

Software Engineering 2: ***Travlendar+***

**Design Document**

Matteo Biasielli - Emilio Capo - Mattia Di Fatta

v. 0.2

**Table of Contents**

**1. Introduction**

1.1. Document purpose..………………………………………………...………………….3

1.2. Definitions, Acronyms, Abbreviations………………………………………….3

1.2.1 Definitions……………………………………………………………………….3

1.2.2 Acronyms………………………………………………………………………..3

1.2.3 Abbreviations…………………………………………………………………..3

1.3. Reference Documents…………………………………………………………………4

1.4. Document Structure……………………………………………………………………5

1.5. Revision History…………………………………………………………………………5

**2. Architectural Design**

**3. Algorithm Design**

3.1. Best Travel Option

3.1.1 Natural Language Description

3.2. Overlapping New Activity Check

3.2.1 Natural Language Description

3.2.2 Java Code

**4. User Interface Design**

4.1 Additional UIs

4.2 Mapping UIs into Requirements

**5. Requirements Traceability**

**6. Implementation, integration and test plan**

**7. Effort Spent**

**1. Introduction**

* 1. **Document purpose**

This document has to be intended as a general guide for the correct development of the Travlendar+ application. The content of this document follows and is based on the content of the RASD document. This document is meant to be a reference for any person who has an interest in the project. This includes, but is not limited to, development team members, stakeholders and end users.

* 1. **Definitions, Acronyms, Abbreviations**
     1. **Definitions**
* **User**: actor that is using the application and may want to access all functionalities.
* **Application**: with the term application we are talking about the desktop version, the website and mobile version of the Travlendar+ system.
* **Scheduling**: action performed by a user that is adding a new activity to his personal calendar.
* **Flexible Activity**: An activity with starting and ending time larger than the duration.
* **Fixed Activity**: An activity with fixed starting and ending time.
  + 1. **Acronyms**
* **RASD:** Requirements Analysis and Specification Document
* **DD:** Design Document
* **UI:** User Interface
* **API:** [Application programming interface](https://en.wikipedia.org/wiki/Application_programming_interface)
* **UXD**: User Experience Diagram
* **UML**: Unified Modeling Language
* **GPS**: Global Positioning System
* **DECS**: Dynamic Event Check System
  + 1. **Abbreviations**
* **[Gn]:** the n-th goal
* **[Rn]:** the n-th requirement
* **[NFRn]:** the n-th non-functional requirement
* **[An]:** the n-th assumption
* **[Cn]:** the n-th constraint
* **[UIn**]: the n-th user interface example
  1. **Reference Documents**
* Mandatory project assignments for the A.Y. 2017/2018 available on the beep’s page of the Software Engineering 2 course.
* Projects examples and other documents available on the beep’s page of the Software Engineering 2 course.
* RASD Document available on the Delivery Folder on the repository <https://github.com/MatteoBiasielli/BiasielliCapoDifatta> .
  1. **Document Structure**
* **Introduction:** This is the very first part of the document.

In this section it’s possible to retrieve general information about the Design Document. The purpose and intended audience of the document are specified here.

In addition, Acronyms, Definitions and Abbreviations are defined in this section in order to make it easier, more concise and clearer to read the rest of the Design Document.

* **Architectural Design:** This part represents the second chapter of the document. Here the reader can find the architecture of the system components at various levels and contexts.

First of all, a high-level overview of the components and the way they’re connected is provided in this section. Following this, some components will be analysed in detail and their internal architectures will be showed for a matter of clarity.

* **Algorithm Design:** The most important algorithms that will be implemented in our application are described here, both with natural language and with java code/pseudocode.
* **User Interface Design:** Some User Interface samples have already been provided in the RASD document but they’ll be extended and some will be added in this section of the Design Document.

In addition, further explanation about the already existing UI will be added here, together with a detailed mapping of the User Interfaces into functional requirements and non-functional requirements.

* **Requirements Traceability:** Design choices are mapped into functional and non-functional requirements here.
  1. **Revision History**
* **v. 0.1 [28 Oct 2017]**: added the whole “Introduction” section.
* **v. 0.2 [06 Nov 2017]:** Added “Algorithm Design” and “User Interface Design” sections.
* **v. 0.3 [07 Nov 2017]:** Added part of Architectural design

**2. Architectural Design**

In this section, we first provide a general overview on Travlendar+

system by means of a general Component Diagram. Then it is given a Component View on some important components again by means of Component Diagrams.

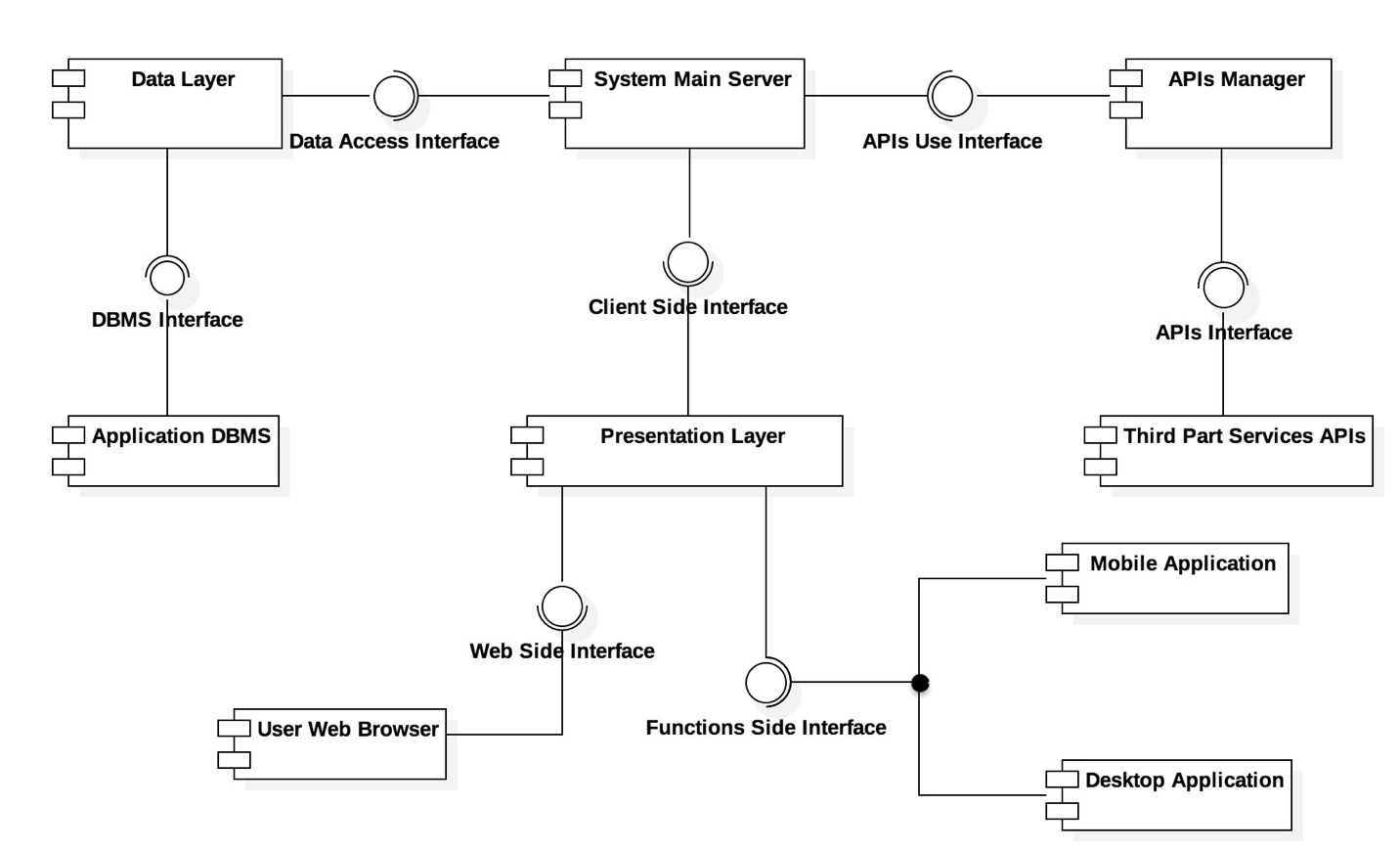
Moreover, a Deployment View and a Runtime View of the system is provided in order to describe nodes with their components, protocol for their interaction and how they are expected to work (by means of Sequence Diagrams).

Eventually, we describe in detail all components’ interfaces that can be found in the general Component Diagram.

**2.1 Convention**

**2.2 System Overview**

The following diagram describes the whole Travlendar+ system with all its components (i.e. software modules).



***Data Layer***

This component deals directly with system’s DBMSs allowing other components to write, read and update data in the DBs properly (i.e. in a secure, consistent way) using the proper DBMS. It provides a single interface outward in order to systematize access to data stored in DBs and hide the internal complexity and implementation of DBMSs and DBs.

***Application DBMSs***

This particular component is used as generalization of the two DBMSs used in Travlendar+: one to manage users’ personal data and calendar, one to manage all relevant internal information used by Travlendar’s server to works properly (for further information see Component View section).

It provides an appropriate interface to allow the Data Layer component, and only it, to query DBMSs.

***System Main Server***

This is the core component of the server side of the system and of the application in general. It’s composed by three essential components: the Computation Unit, the User Side Unit and the Dynamic Event Check System. It manages, with the aid of the APIs Manager and the Data Layer, all the Travlendar’s functionalities described in the RASD document, except for the presentation side, managed entirely by the Presentation Layer (see below for further information).

***APIs Manager***

This component is used to homogenize the different kinds of APIs provided by third part services, in other words it adapts the external APIs to the system in order to make them easily usable. By doing this, the APIs Manager tries also to optimize the access and the usage to the APIs by means of an internal optimizer component. It provided the System Main Server with the APIs Use Interface, used to exploit third part services while masking their implementation.

***Third Part Services APIs***

This component is just a collection of all needed APIs with their internal representation, tools and external references.

***Presentation Layer***

The Presentation Layer component is the one appointed to show the front-end of the Travlendar+ system. Making use of the Client Side Interface provided by the System Main Server, it grants access to system’s data to users w.r.t. confidentiality (i.e. only authorized users/clients will have access to data they’re allowed to access and that are made accessible) by means of a website and the mobile and desktop application.

It provides two different interfaces, the Web Side Interface and the Functions Side Interface, as the user can interact in two ways with the system: through the website where he can download the application, find useful information (e.g. FAQs) and additional documentation, or through the desktop/mobile application by means of which he can exploit Travlendar+ functionalities.

***Website***

This component models the website page of the project where a user can download both the desktop and the mobile version of the application and where he can find support documentation provided by developers themselves. It makes use of the Web Side Interface to connect to the system.

***Mobile Application***

The Mobile Application component represents an abstraction of the mobile version of Travlendar+, used in this diagram to show its interaction with the whole system. It makes use of the Functions Side Interface to connect to the system.

The core function of this component is to keep track of the itineraries contained in the travel options provided by the Computation Unit in the System Main Server in order to send notification to the user and notify him of possible/critical changes in their travel options.

***Desktop Application***

The Desktop Application component represents an abstraction of the desktop version of Travlendar+, used in this diagram to show its interaction with the whole system. It makes use of the Functions Side Interface to connect to the system.

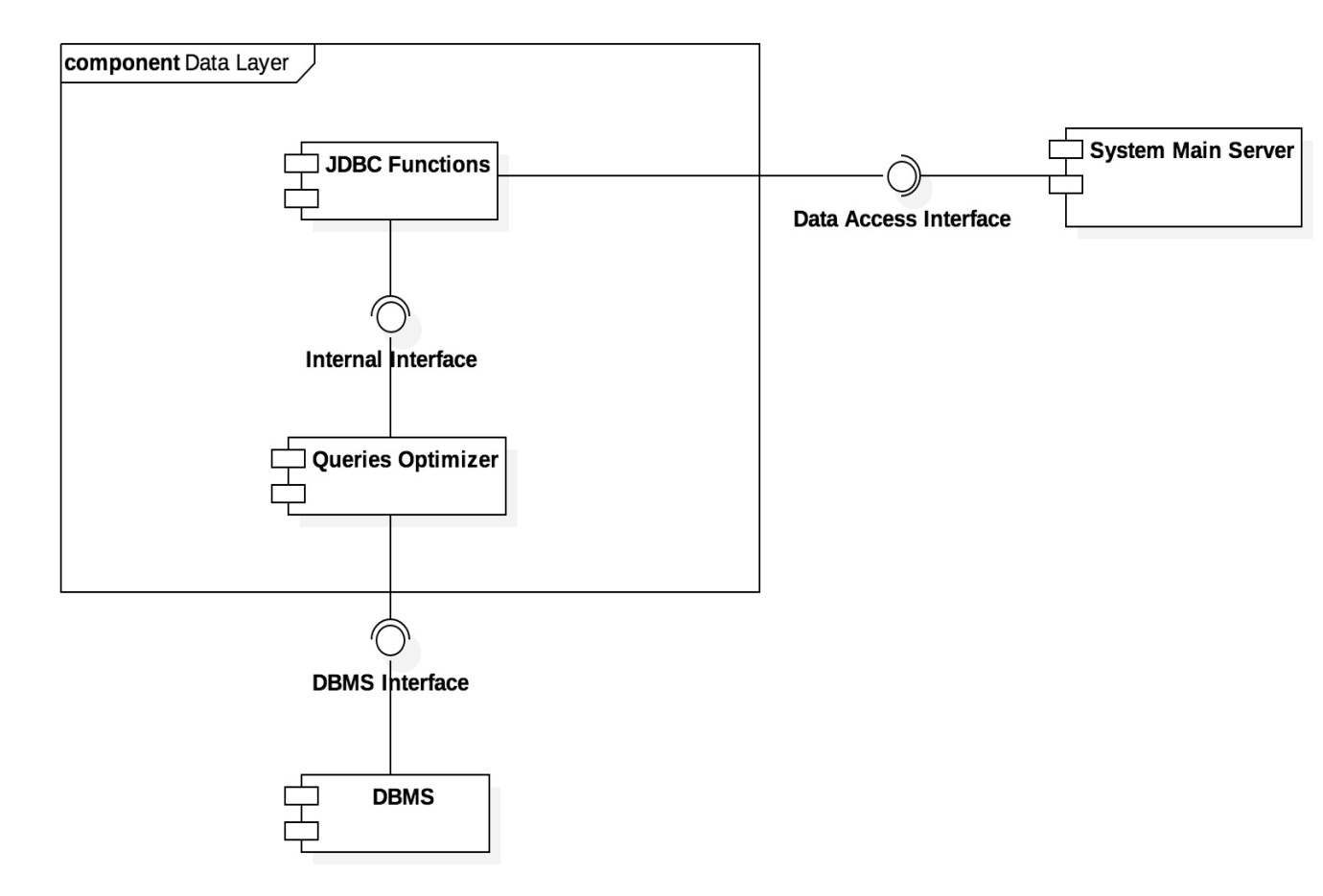
The core function of this component is to keep track of the itineraries contained in the travel options provided by the Computation Unit in the System Main Server in order to send notification to the user and notify him of possible/critical changes in their travel options.

***User Web Browser***

This component models the web browser used by the user to access the website. It’s an external object with respect to our system, but its corresponding component is necessary in order to model its interactions with the website.

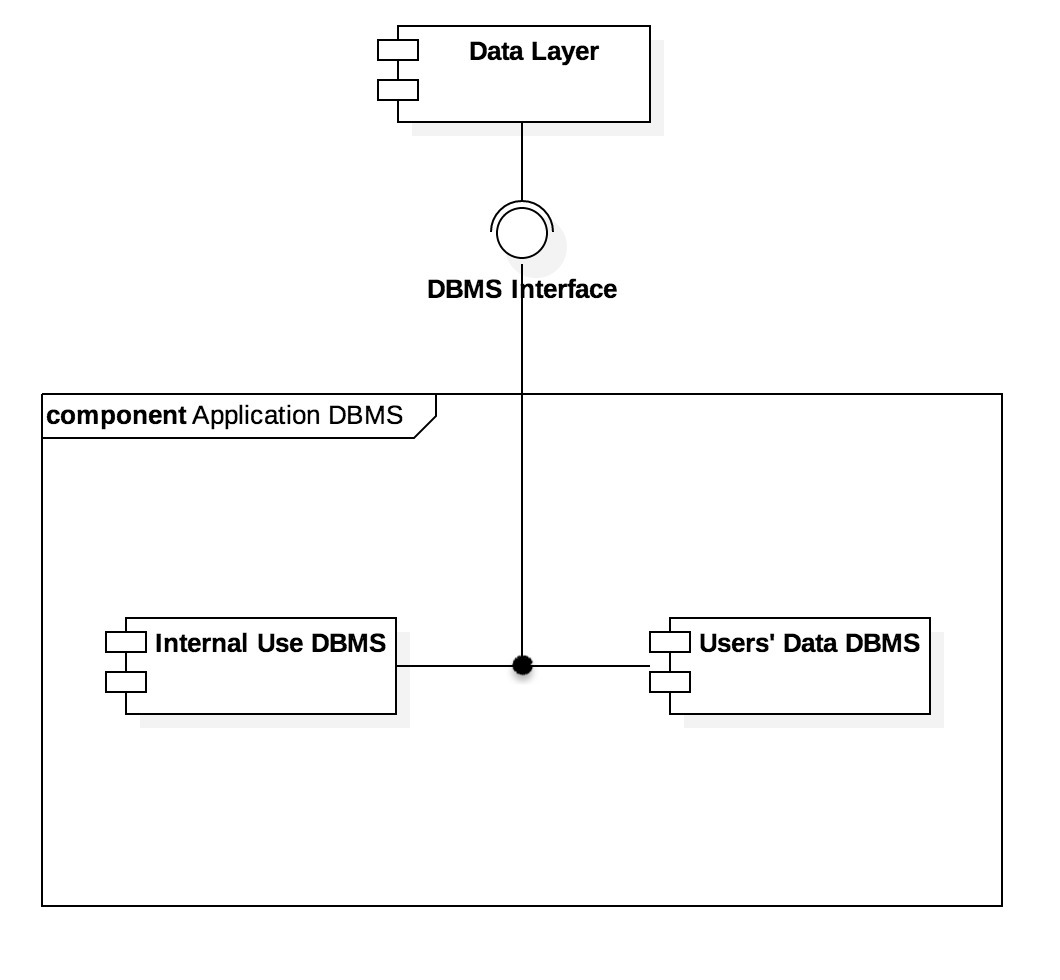
***2.3* ComponentView**

***Data Layer***

******

The Data Layer module is composed by two sub-components: the JDBC Functions Component and the Queries Optimizer Component. The former is a collection of tools and references used to integrate and use the JDBC library (Java library for DBs handling). The latter is used to optimize the formulation of queries by means of JDBC tools.

***Application Database Management Systems***

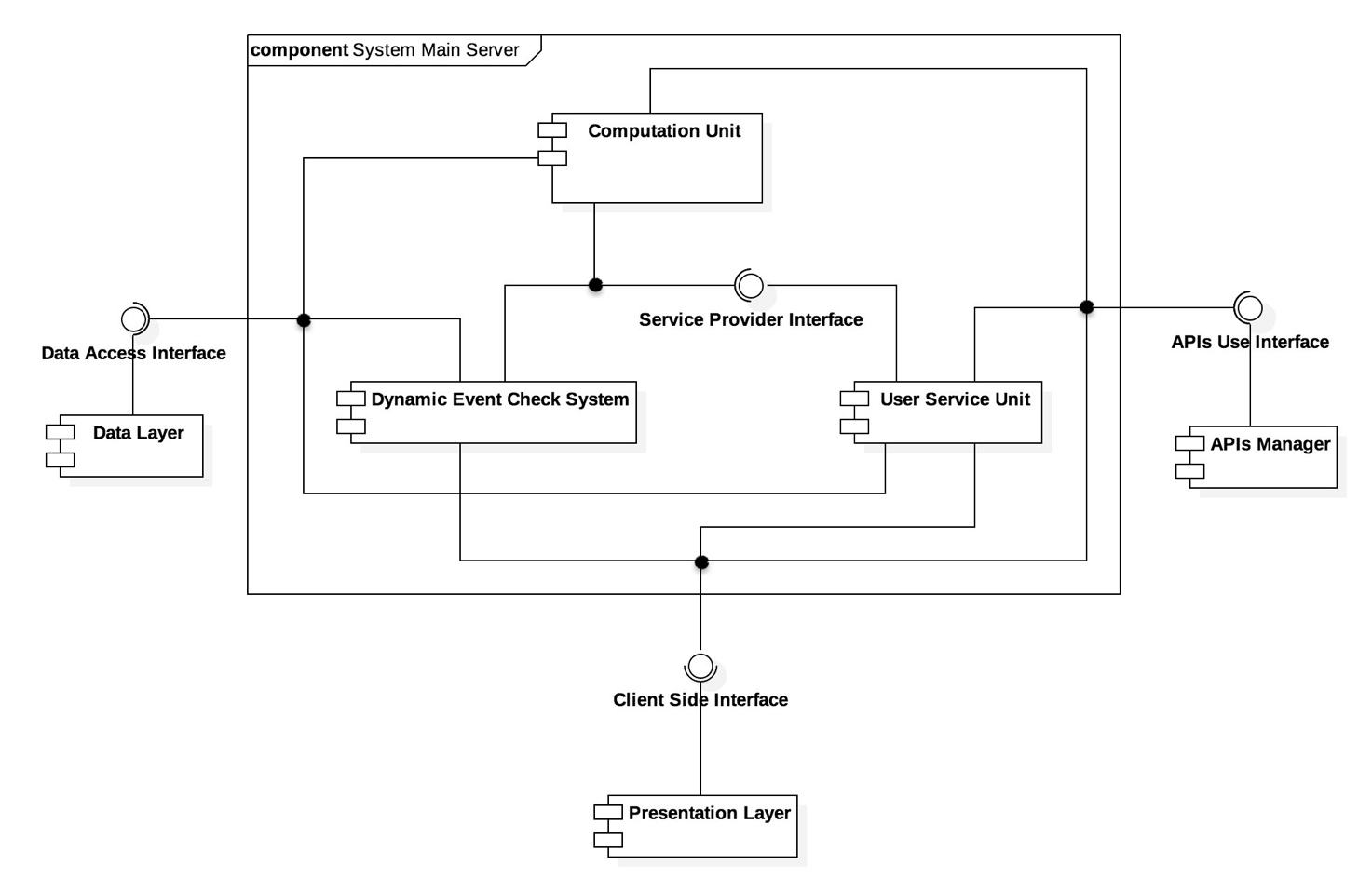
******

The Application DBMSs, as mentioned in the previous section, is a generalization of the two DBMSs used in the Travlendar+ system.

The Internal Use DBMS component models the DBMS that deals with the DB in which the system stores all the data needed to make the Travlendar+ application and server works properly.

The Users’ Data DBMS component models the DBMS used query the DB in which the system stores users’ credentials, calendars and sensitive data. All these data are encrypted.

***System Main Server***

******

The System Main Server is the core component of our server. It acts as “brain” for Travlendar+ since it must compute the best travel option when needed for each user (by means of the its Computation Unit), check dynamically (i.e. when events happen it’s listening) for events which can change travel options for some users and reports these to the local application(by means of the Dynamic Event Check System) and provide all these data (and some more) to the user in a proper way (by means of the User Service Unit).

Since this component has to be scalable and high-performing we could decide to distribute this component over several physical machines in case of necessity.

The communication between its component is allowed by the Service Provider Interface, an internal interface provided by the User Service Unit to transmit data outwards.

All the three internal components makes use of the Data Access Interface through which they can query DBs properly and of the APIs Use Interface through which they can exploits third part services.

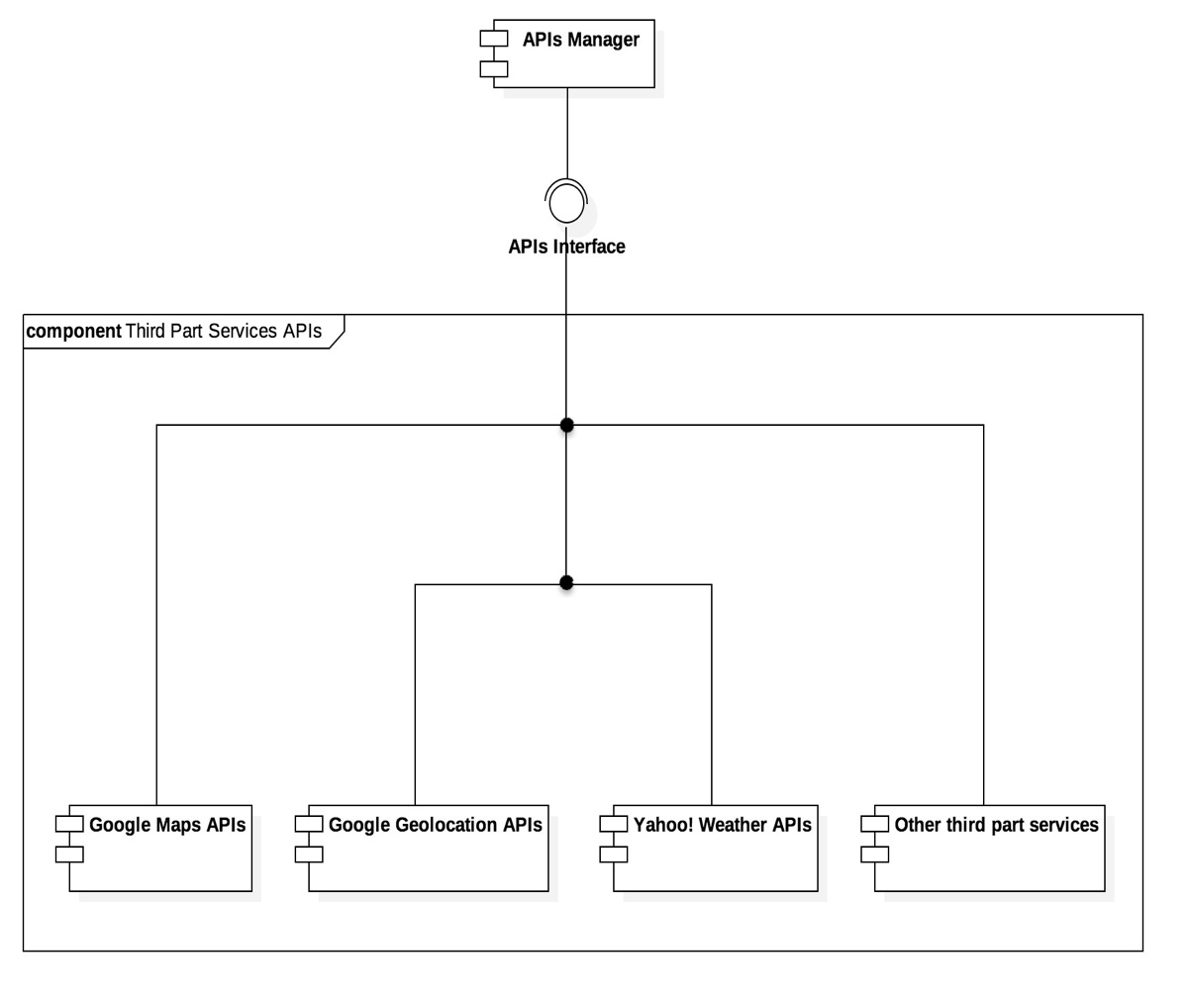
***APIs Manager***



The APIs Manager is internally composed by two components and an interface to let them communicate.

The Adapter component is used to make homogenous all different kinds of APIs exploited by the system by encapsulating them according to OO principles for sake of clarity and order. It also provides the Internal Interface to let other components make use of these APIs.

Moreover, the Optimizer tries to optimize the access to APIs and the computation in using them.

***Third Part Services APIs***

This component is a container of all commercial partners’ APIs. The system makes use of the OO principle of ‘encapsulation’ to efficiently and methodically have a reference to all these external tools. This way, this component of our system is easily extensible.

**3. Algorithm Design**

In this section, the most important algorithms, that will need to be implemented in order to make our application work properly and satisfy the goals, will be described with details.

* 1. **Best Travel Option**
     1. **Natural Language Description**

The problem is: given a user, a starting point and an arrival point, the algorithm has to compute the best travel option that leads from the starting point to the ending point and that satisfies the preferences expressed by the user.

The system running the algorithm has, thanks to the assumptions made in the RASD document, access to all the APIs of the third part systems involved:

* + - Google Maps;
    - Car Sharing Systems;
    - Bike Sharing Systems;
    - Yahoo! Weather.

The access to APIs is granted through the API manager (see class diagram), so it’s fair to group them by category, as listed above.

In order to avoid useless calculations and delays, the Algorithm first checks the user’s preferences to check which vehicles the user has declared as available and which Vehicle Sharing services he agreed to take into consideration.

The list of available vehicles is modified as follows:

* + - If the starting point corresponds to the user’s home, then the list is not modified.
    - Otherwise, the user is asked which of the vehicles that he owns are available (that should be the one he used to go out before or none if he went out using another transport) at the moment and the list is restricted. If the user does not reply within 30 seconds, the list is considered empty.

Then, the weather is checked through the API manager and the lists are then restricted again as follows:

* + - If the user is home and forecasts say it’s going to rain in a moment the user should be travelling, then bike is removed from both lists;
    - If the user is not home and forecasts say it’s going to rain then bike sharing is removed from the list of Vehicle Sharing systems available(if present);
    - Otherwise, the lists are left untouched.

Now, the following calculations are performed:

1. For each vehicle in the available vehicle’s list, through the API manager, the travel option from the starting position to the ending position using the vehicle is computed.
2. For each category of Vehicle Sharing system available, the nearest one is located. Then, for each of them, the best travel option corresponds to the best travel option to reach the vehicle plus the best travel option from the vehicle’s position to the destination (using the vehicle). In order to have a consistent final travel option, both sub-travel options must satisfy preferences.
3. The best travel option using public transport (and on foot) are computed.

Then:

* + - For each travel option in the list 1, if the user is home, considering that if he goes out with a certain vehicle he may want to use it until he goes home again, it is checked is a good travel option using that vehicle is available among all other events of the day. If not, the travel option is removed from list 1.
    - If the user is not home and list 1 contains a travel option, all other lists are emptied because he will need to use his vehicle.

Following, the travel options that don’t satisfy the user’s preferences are removed from the lists. At this very moment, if the user is using any special modality (e.g. minimize cost modality), it is verified correctly, leaving in the lists only the travel means that satisfy them. Furthermore, travel options are deleted also if they take more than the available time.

The last step is:

* If at least one list is not empty, the travel option that takes less time is taken from each list and presented to the user.
* If all the lists are empty, it means that the user can’t arrive to the place on time and have his preferences satisfied at the same time. In this case, the user is presented the travel options as they were before the application of the preferences and the user is warned of the situation.
  1. **Overlapping New Activity Check**
     1. **Natural Language Description**

The problem is: given a user and an activity, the algorithm has to check if the activity can be added to the calendar.

A first check has to be done on the activity:

* + - If the activity to be added is a fixed activity, then:
      * For each **fixed** activity Act already present in the user’s calendar, if Act has either one or both starting and ending moment strictly included in the time span defined by the starting and ending moment of activity to be added, return **false**.

This guarantees that there’s no overlapping with fixed activities;

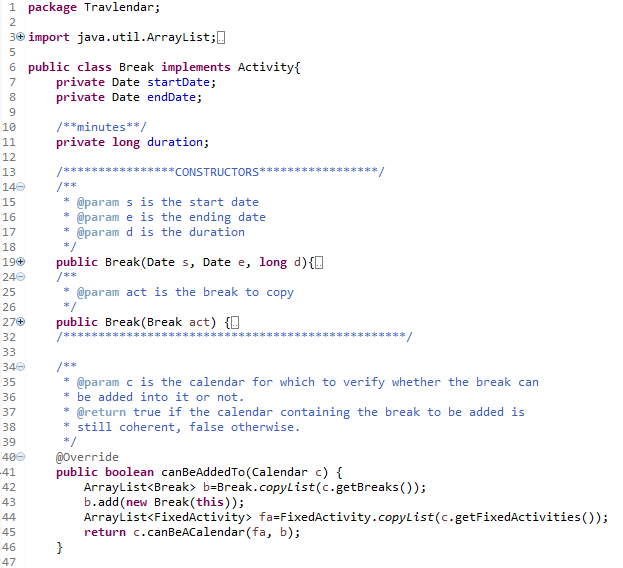
* + - * If the algorithm got to this point, it means that the activity to be added does not overlap with other fixed activities.

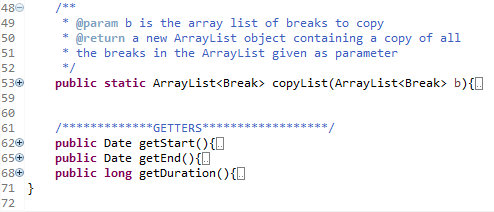
Considering now a calendar that contains also the activity to be added, if at least one of the flexible activities doesn’t have anymore a possible “placement” for the effective activity duration in the range defined by the starting and ending moment, return **false**;

* + - * Return **true** if false has not been returned in the previous checks;
    - Otherwise, it means that the activity to be added is a flexible activity. In this case, if there is a possible “placement” for the effective activity duration in the range defined by the starting and ending moment and all other flexible activities can still be placed, return **true**. Otherwise, return **false**.
    1. **Java Code**

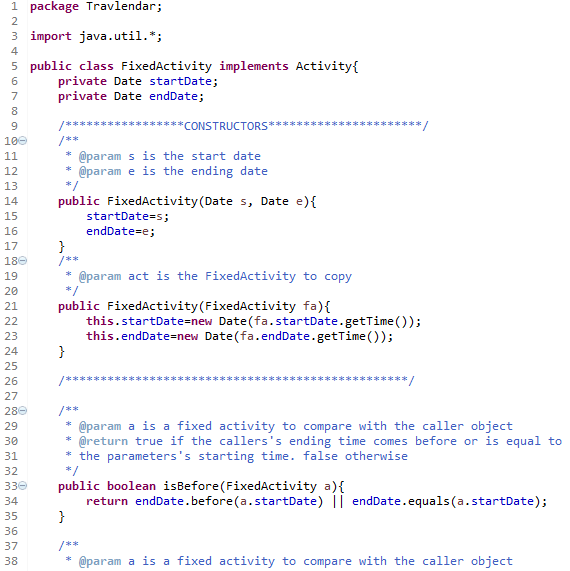
In this section, only strictly necessary methods are represented. Furher information and the complete code can be found in the attachments and/or in the delivery folder.

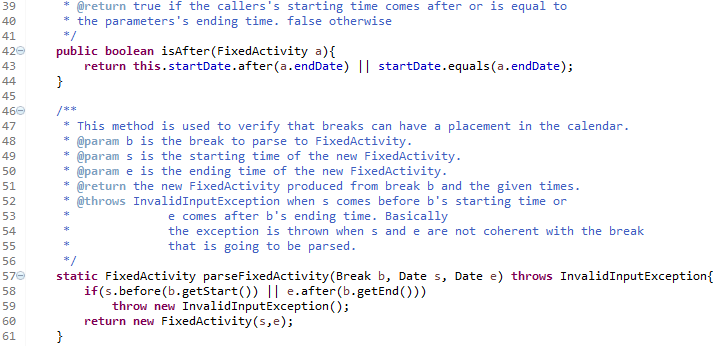
* + - **Break class**

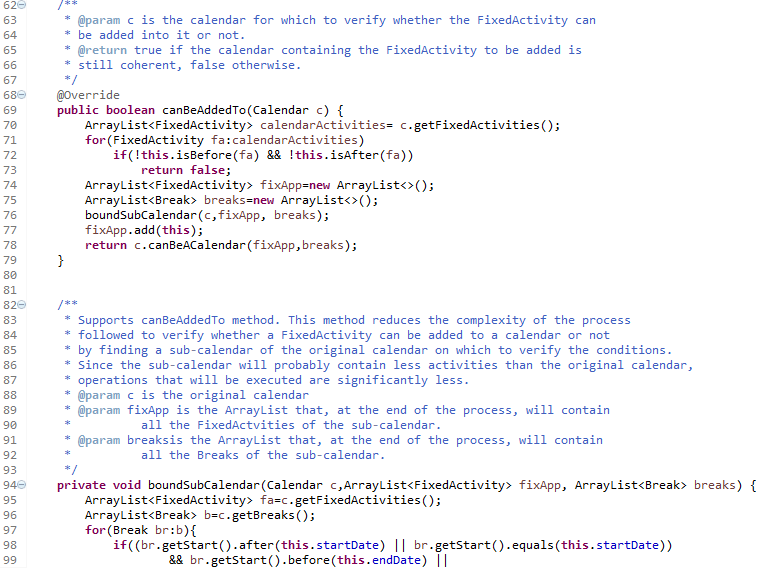


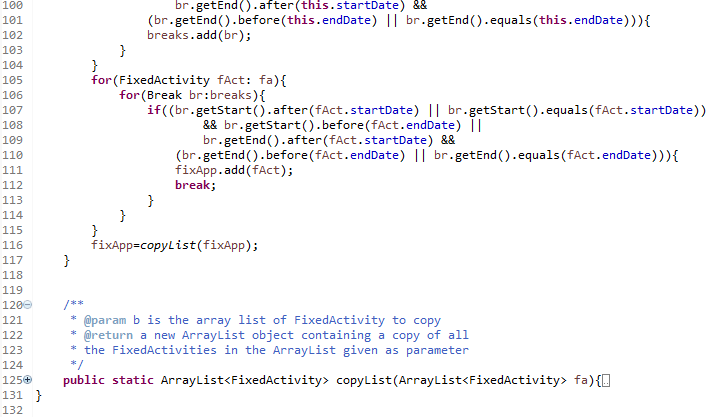


* + - **FixedActivity class**

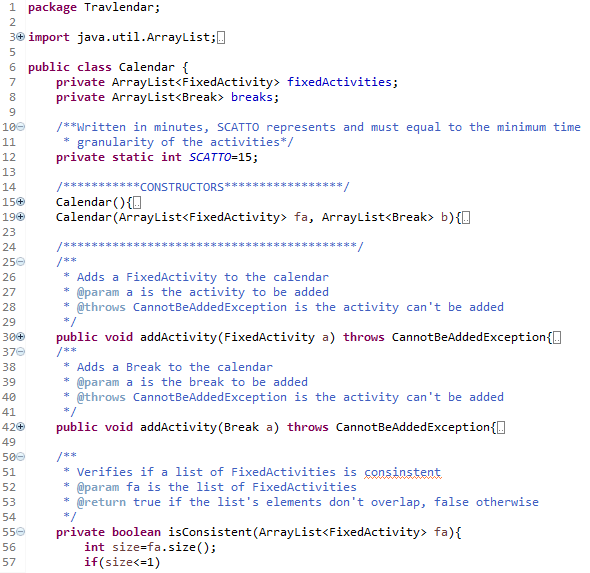


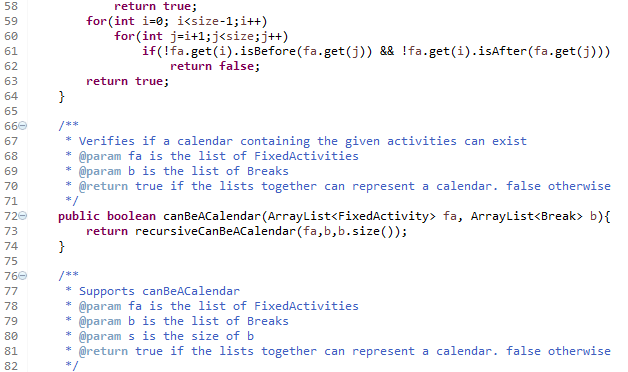






* + - **Calendar class**





Note that this small example has been provided with the only purpose to show the structure of the algorithm (that actually works because this is a complete code). Some previous or further operations that are not strictly related to the execution of the algorithm (e.g. load activities data from a DataBase or storing the new Activity in the DB after adding it to a calendar) have not been represented .

**4. User Interface Design**

**4.1 Additional UIs**

Though some UI samples have already been provided in the RASD document ( see from [UI1] to [UI8] in RASD), for a better mapping of our design choices into the requirements another one is provided in the DD document.

**[UI9]**

****

This UI shows how users can differentiate between fixed and flexible activities. This is obtained through a small menu that pops up after tapping on the Create New Activity button.**4.2** **Mapping UIs into Requirements**

Considering the functional requirements, that have already been described in detail in the RASD document, here’s a brief explanation of how our User Interface samples are coherent with the requirements and help us to satisfy them in order to reach the defined goals ( -> RASD ):

* + - [UI4] (and [UI9]) show the day’s activities. By tapping on them it’s possible to reach the screen from which it’s possible to modify an activity. They match with [R1];
    - [UI1], the login screen, matches with [R2]. Furthermore, it also matches with [R3] and [R4] because it shows it’s possible to change password and to register in the system as a new user;
    - [UI8] and [UI9] match with [R5] because they show it’s possible to create fixed and flexible activities;
    - [UI2] satisfies [R6], [R6.1] and [R9] because it shows some of the possible preferences, including the possibility to take into consideration bike and car sharing;
    - [UI6] and [UI7] obviously match with [R11] because they show mobility options and travelling phase.

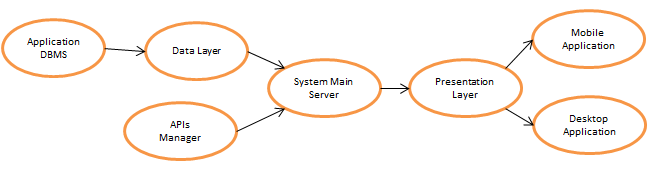
**6. Implementation, integration and test plan**

Considering the diagram showed in the **2.2 System Overview** section as reference, it’s easy to notice which components (depend) on other components. There are components, anyway, that are independent. Considering also that there are no circular dependencies, we can conclude that we are able to implement components according to dependencies, that means that a compoment is implementable if and only if it does not depend on other components or its dependencies have already been implemented. Let’s analyse dependencies:

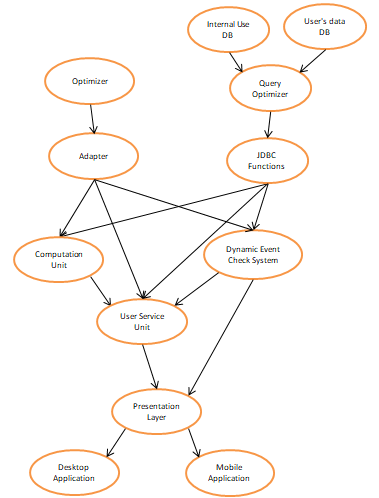
* **Application DBMS** is independent;
* **Data Layer** depends on **Application DBMS**, since it has to build queries and send them to the **Application DBMS** component;
* **System Main Server** depends on the **Data Layer** and **APIs Manager**, since it needs both of them to perform its operations (controller);
* **APIs Manager** relies on the **Third Part Services APIs** in the sense that it can accomplish its role if and only if the **Third Part Services APIs** work fine. However, we don’t need to implement the **Third Part Service APIs** because they have obviously been implemented already by the respective owners, so we can consider the **APIs Manager** independent;
* **Presentation Layer** relies on the **System Main Server**;
* **Mobile Application** relies on the **Presentation Layer**;
* **Desktop Application** relies on the **Presentation Layer**.

For example, a feasible developing order becomes: Application DBMS -> Data Layer -> APIs Manager -> System Main Server -> Presentation Layer -> Mobile Application -> Desktop Application.

Note that the browser is not represented here because it doesn’t need to be developed.

As a matter of clarity, and to highlight other possible orders, we can represent dependencies as a graph, where A -> B if and only of B depends on A. 

The same reasonment can be applied considering the Component view for each node of the graph. Expanding all nodes of the previous graph, we obtain a new and more detailed graph that expresses priorities, showed below.



Considering the way we designed the implementation process, it appears natural to proceed to the integration and test phases with this strategy:

* When the **Application Database** and the **Data Layer** have been completely projected, the **Data Layer** must be completely tested, together with all the queries it exposes to query the Database. Since the **Data Layer** all and only the queries that are useful to the system itself, this will also help to understand if the DB has been projected and implemented correctly;
* Similarly to the Data Layer, the **APIs Manager** exposes all the calls that the system needs to do to external systems. Provided that the Third Part Systems are reachable ,it needs to be tested completely, in order to be sure that the calls are formatted and implemented correctly and that the system receives the results it needs;
* Since the **System Main Server** is implemented after the **Data Layer** and the **APIs Manager** are fully implemented and tested (so we can say they are perfectly functional), it can be integrated with them from the beginning. Moreover, the **System Main Server** should be tested in all its functionalities.
* For the same reasons, the **Presentation Layer** can be integrated with the **System Main Server** immediately.
* When the clients (**Mobile Application** and **Desktop Application**) are developed, they just need to be integrated with the whole system, that is already implemented, integrated and tested and thus can be considered fully and perfectly functional.

**7. Effort Spent**

This section will provide detailed information about the number of hours spent on this document.

**Matteo Biasielli**, matr. 893590

|  |  |
| --- | --- |
| **Section(s)** | **Number of hours** |
| 28-oct-17 Introduction | 1 |
| 29-oct-17 Algorithm Design | 3 |
| 31-oct-17 Algorithm 2 Pseudocode | 1 |
| 01-nov-17 Algorithm 2 Pseudocode | 2 |
| 02-nov-17 Algorithm 2 Pseudocode | 2 |
| 02-nov-17 UI design | 1 |
| 05-nov-17 Google Directions Api | 1.5 |
| 06-nov-17 Review | 1 |
| 05-nov-17 Google Directions Api | 1 |
| 07-nov-17 Group Review | 1 |
| 09-nov-17 Data Layer Implementation | 5 |
| 11-nov-17 Data Layer Implementation | 5 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| **TOTAL:** |  |

**Mattia Di Fatta,** matr. 893608

|  |  |
| --- | --- |
| **Section(s)** | **Number of hours** |
| 28-oct-17 System Component Diagram | 1 |
| 29-oct-17 System Component Diagram + single Component Diagram | 2.5 |
| 30-oct-17 single Component Diagram | 2 |
| 31-oct-17 Started Descriptions of Component Diagrams | 2 |
| 1-nov-17 Descriptions of Component Diagrams | 2 |
| 4-nov-17 Descriptions of Component Diagrams | 2 |
| 6-nov-17 started component interfaces | 1.5 |
| 07-nov-17 Group Review | 1 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| **TOTAL:** |  |

**Emilio Capo,** matr. 899842

|  |  |
| --- | --- |
| **Section(s)** | **Number of hours** |
| 4-nov-17 Runtime View | 1.5 |
| 5-nov-17 Runtime View | 1.5 |
| 07-nov-17 Group Review | 1 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| **TOTAL:** |  |