

Politecnico di Milano

A.Y. 2017/2018

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

**Design Document**

Matteo Biasielli - Emilio Capo - Mattia Di Fatta

v. 0.1

**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

3.1.2 PseudoCode

3.2. Overlapping New Activity Check

3.2.1 Natural Language Description

3.2.2 Java Code

**4. User Interface Design**

**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
  + 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.
  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.

**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. **PseudoCode**
  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;

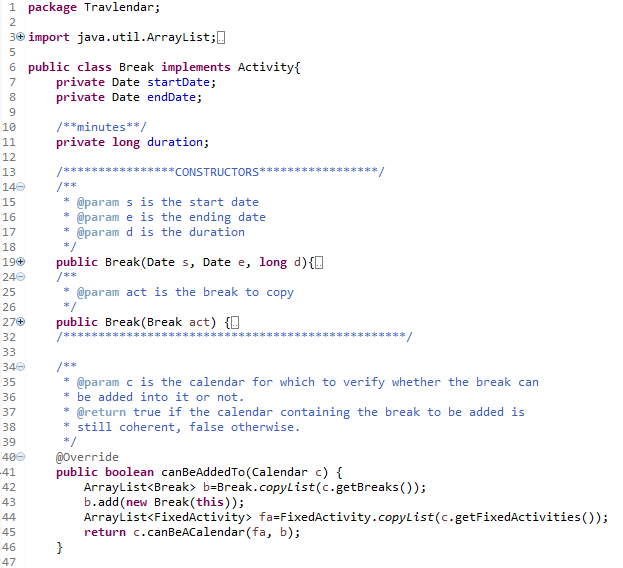
* + - * If the algorithm got to this point, it means that the activity to be added does not overlap with other fixed activities.

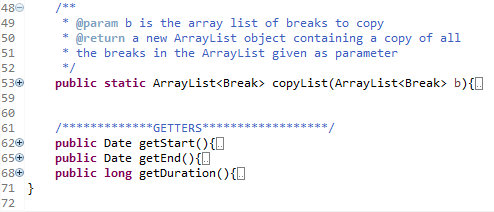
Considering now a calendar that contains also the activity to be added, if at least one of the flexible activities doesn’t have anymore a possible “placement” for the effective activity duration in the range defined by the starting and ending moment, return **false**;

* + - * 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**

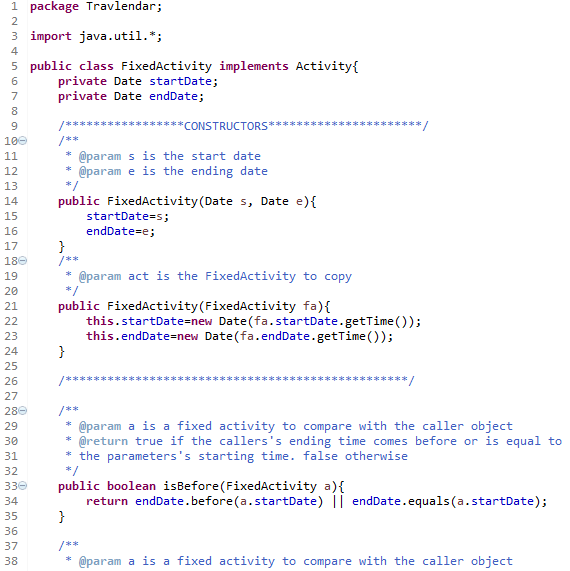
In this section, only strictly necessary methods are represented. Furher information and the complete code can be found in the attachments and/or in the delivery folder.

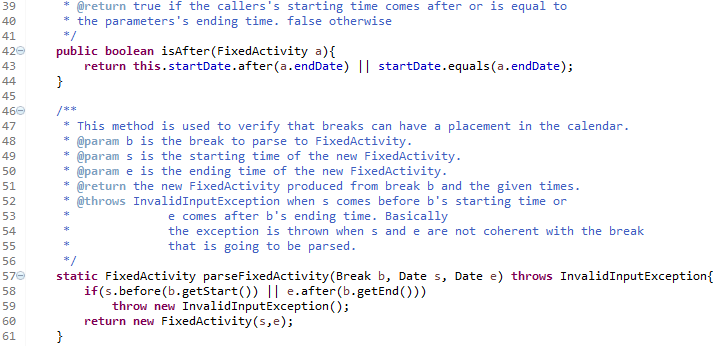
* + - **Break class**

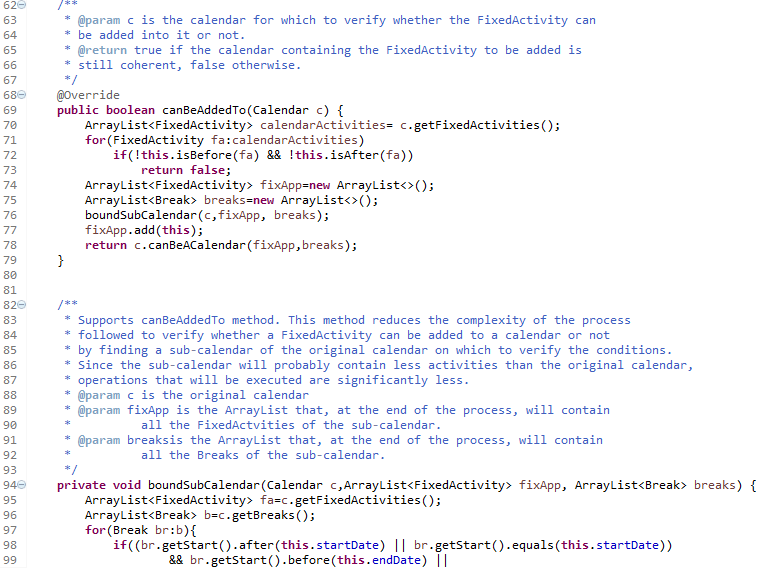


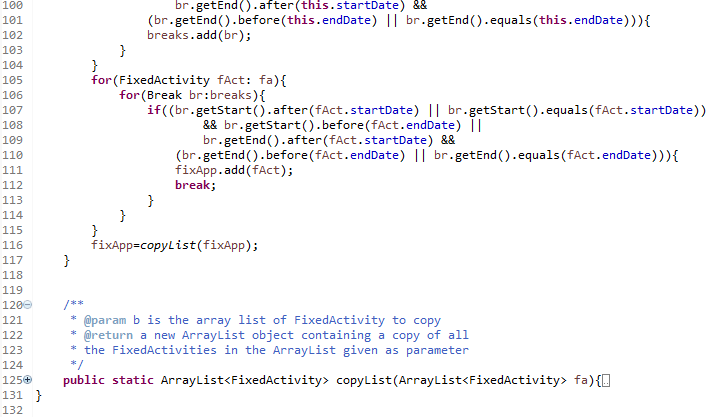


* + - **FixedActivity class**

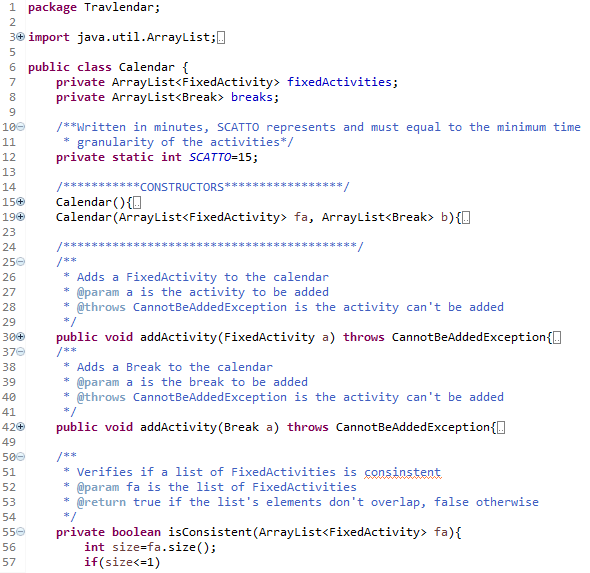


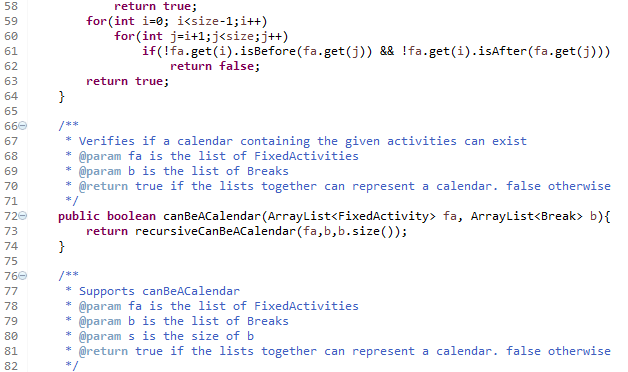






* + - **Calendar class**





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 .

**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 |
| 1-nov-17 Algorithm 2 Pseudocode | 2 |
| 2-nov-17 Algorithm 2 Pseudocode | 2 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| **TOTAL:** |  |

**Mattia Di Fatta,** matr. 893608

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| **TOTAL:** |  |

**Emilio Capo,** matr. 899842

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| **TOTAL:** |  |