Manuel Benavent Lledo, W1749459

Enterprise Application Development CW2 Report

This report will focus on the design changes and their reasons from coursework 1 and the maintainability of the software after the implementation of this second coursework.

Firstly, we will start discussing about the design modifications from coursework 1. All the requirements have been accomplished since they describe program functionality and they have nothing to do with the implementation decisions. The data base described in the requirements has been implemented as designed, adding a datetime attribute to the Event table. The implementation for the contact-event has one table per event type since the event tables are implemented as one per type so the access is easier (simpler SQL statements) and faster (only need to access 1 table to retrieve the data for an specific event).

The second part from coursework 1 was the use case diagrams, since they describe briefly what the systems needs to do, all of them are implemented as designed. When it comes to the third part of the design document, the classes, there have been several changes in the implementation but maintaining the functionalities. The most important and remarkable is the Data Access Component classes (DAC), it is very easy to realize once the implementation has started that it is not efficient to have a Data Access Component class for each model class since there is no code reusability at all and therefore the maintainability is horrible. However, it is not efficient either to include this code in each model for the same reason and, the data base accesses would be distributed all around the model classes which would not meet the layered approach described in the design and would lead to huge maintainability issues that will be described later when it comes to using a different data base. So, the implementation decision was to include a unique Data Access Component class with methods as generic as possible so in case some changes need to be performed, only 1 file will have to be modified.

Another remarkable implementation variation is that there are not as many exceptions as expected while designing. Instead of validating the data and throwing an exception if it is not correct, the data is validated in the controller and it will only create the object if the data is correct. Nevertheless, the exception *NoDataException* has been kept for the time report algoritithm and a new one for DAC errors (*DDBBException*) has been added during the implementation. The last modification for the classes part is the ICRUD interface, since there are not several Data Access Component classes anymore this interface has been deleted. However, since all the model classes have some common methods that will be used by the DAC class, in order to have more generic methods, the interface *ISQLAccess* has been implemented.

For the third part, the sequence diagrams, the only modification takes place where the validations of the details happen, as explained before, in the controller instead of the model. And the fourth and last part, the activity diagram describing the time report algorithm, it has been implemented as designed except for the last part, instead of calculating an average value per week, the regression equation parameters are calculated and used to predict the new values based on the input data for the equation (the 4 previous weeks).

The second part of this report is related to the maintainability of the code, all the implemented methods and classes have been implemented so that the code can be reused and there is no access to data from classes that should not have access to certain areas of the code. Starting from the solution, it contains 2 different projects, library is a class library that is used by the main project in order to keep abstraction. The main project can access all the public methods in this library, but their implementation is separated. Moreover, if we want to implement any other C# project, such as a web using ASP.NET or an iOS/Android app using Xamarin, they can use the exact same model library with some minor modifications related to the DDBB connection string since it works differently depending on the final platform.

In the library project, we have all the classes that are designed to be part of the model in the MVC pattern, all of them are public except for the DAC which is internal since the controllers or views should not access the data directly. In all the classes properties are used instead of getters and setters which will improve the code readability. The most remarkable part for the model implementation when it comes to maintainability of the library project are the different methods in the DAC class, for creating, updating or deleting in the DDBB there are only 4 methods for the all the model classes and if we added more classes it will remain, the Create, Delete and Update methods will generate a SQL statement based on the type of object and some of its properties (all the classes except for the DAC in the model implement the interface *ISQLAccess* so that the properties are the same for all of them) and the fourth method will execute the SQL non query statement which would be a common code part for the other 3 methods, if we need to fix anything related to one of this SQL actions only 1 method will have to be modified (as said previously, if we had had a DAC class per object or the data base access had been implemented directly in the model, all this code that is being reused would have been copied and pasted in different places so if modifications needed to be carried out it would take the time for the DAC times the number of different model classes that we have). Even tough the reading DAC methods might not have as good maintainability since we would have to change each method, where it has been possible the method has been parameterized so it can be used by different functions that require similar actions with the same return type (e.g. the *ListEvents* method includes the parameter for the dates range).

Another maintainability detail from the library project is the abstract class for event, since all the event types will have the same basics, they are implemented in the *EventClass* class. This is an abstract class which is the basic for any new event type that we might include. It has some abstract methods that all subclasses must implement, non-abstract methods that are shared by all the subclasses improving the maintainability and due to the abstract class, we make sure that all the events have a type since it cannot be instantiated.

For the main project, it is not possible to apply that much code reusability except for the overloaded constructors in some of the forms to have different views depending if we are creating or updating. Despite the "bad" code reusability, the maintainability is still good since the partial classes of C# allow us to divide the static view from the controller and the dynamic view. So, the autogenerated view code is in one file and the rest in the other file. Different private methods have been defined per class, then the modifications only need to take place at a single spot.