Fight Club Planner

Analysis and Design Document

Student: Stancu Gabriel - Iulian

**Group: 30432**

Table of Contents

1. Requirements Analysis 3

1.1 Assignment Specification 3

1.2 Functional Requirements 3

1.3 Non-functional Requirements 3

2. Use-Case Model 4

3. System Architectural Design 6

4. UML Sequence Diagrams 8

5. Class Design 9

6. Data Model 10

7. System Testing 11

8. Bibliography 11

1. Requirements Analysis

# Assignment Specification

Design and implement an application that helps MMA tournaments manage their scheduled fights better while ensuring covid safety standards.

# Functional Requirements

A manager should be able to create a tournament and invite fighters to sign up for a venue. Each tournament requires at least bi-weekly matches to generate traction and revenue within the tournament period.

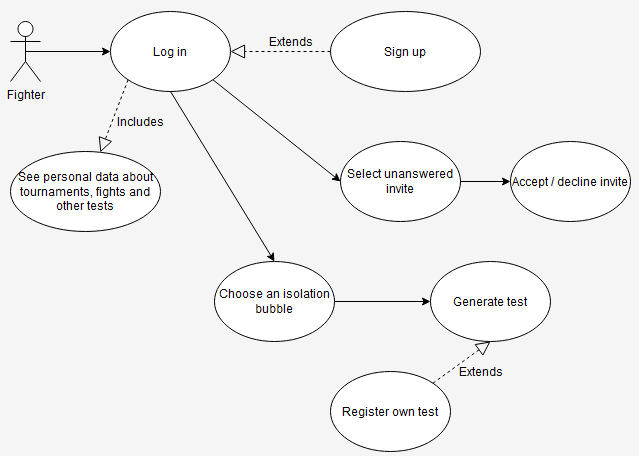
After they sign up, in order to ensure proper safety standards, fighters are required to present themselves at a tournament isolation bubble with a test which will be recorded and they will immediately be tested again on site. If the “arrival” test is positive the fighter is then moved to quarantine until the test results are negative again.

In order for a fighter to take part in a tournament they require at least 3 weeks of negative test history after which they can be matched up with similar fighters of their caliber.

A manager should be able to see in real time the tournament schedule being populated with eligible fighters (at least bi-weekly).

# Non-functional Requirements

* Implement and test the application
* Use an ORM and a DI Container
* Commit the work you do on your Git repository
* Use any OOP language you like. Non-exhaustive: Python, C#, Java, Ruby, C/C++, JS+Typescript
* Use a layered architecture
* Use the builder pattern for creating the tournament schedule
* The data will be stored in a database
* All the inputs of the application will be validated against invalid data before submitting
* the data and saving it in the database.

2. Use-Case Model

**Use case**: Fighter accepts tournament invites

**Level**: user – goal

**Primary actor**: Fighter

**Main success scenario**: the fighter logs in, checks the invites datagrid, clicks on a non – answered one, accepts / declines it

**Extensions**: the fighter is not registered, so he has to sign up first

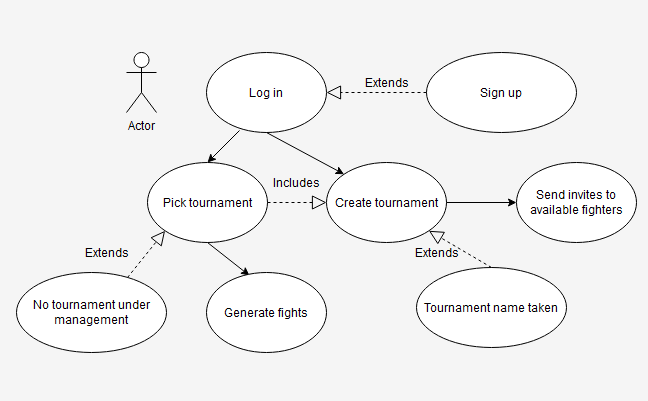
**Use case**: Fighter tests himself

**Level**: sub-function

**Primary actor**: Fighter

**Main success scenario**: the fighter logs in and tests himself for being eligible for fights. In order to achieve this, a fighter should test himself regularly (at least one test per week and all tests negative) for 3 consecutive weeks.

**Extensions**: the fighter is not registered, so he has to sign up first and register a test taken by himself before being tested

****

**Use case**: Manager creates tournaments and invites fighters to it

**Level**: user – goal

**Primary actor**: Manager

**Main success scenario**: the manager logs in, creates a new tournament, then sends invites to available fighters

**Extensions**:

* the manager is not registered, so he has to sign up first.
* the tournament name is taken, so he needs to rename it.

**Use case**: Manager generates fights and sets the winner

**Level**: sub-function

**Primary actor**: Manager

**Main success scenario**: the manager picks a tournament he is responsible for, selects a date for starting the fight week (by default it is the current date) and the system generates fights between the available fighters; after fight has taken place, the manager can pick the winner. The use case assumes the manager will wait until a specified moment to generate the fights, so that the fighters can test themselves and become eligible. If a fighter is found positive or he is not tested regularly, he won’t be considered eligible for fighting.

**Extensions**: the manager has no tournament under management, so he has to create one and invite fighters to it

3. System Architectural Design

**3.1 Architectural Pattern Description**

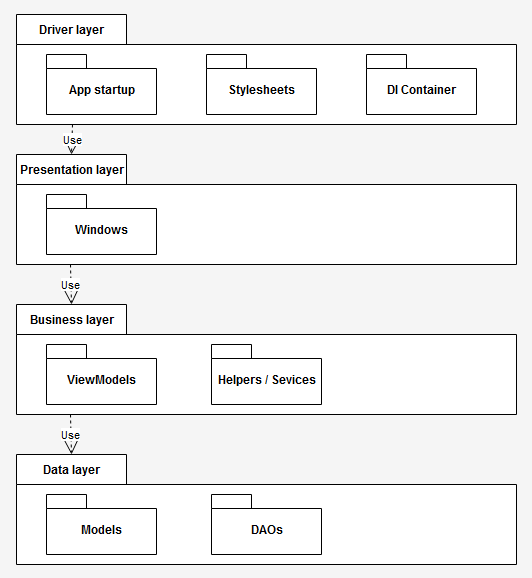
The main architectural pattern used for developing the project is the layered pattern. We divided the application in 4 distinct layers:

* The data layer: the layer responsible for mapping the database entities into program classes; it contains model classes of the entities in the database and database access objects for each model / entity used in the application. At the level of DAOs we define the queries needed to communicate with the database. For mapping between queries results and application objects / classes, we used an ORM framework (EF Core). This layer has no dependency on other layers.
* The business logic layer: the layer responsible for the functionality of our application. This layer is divided in two main groups: the View – Models used by the Presentation layer to show data in the UI and the helper classes, and the helper classes. The helper classes provide actual functionality of the application, working with arguments received from other layers, creating a dependency inversion for this matter. In order to achieve this, a DI container was used (Autofac). This layer only depends on the data layer.
* The presentation layer: it contains the views the user can see and interact with. This layer depends on the business logic layer mostly. However, small dependencies on the data layer had to be provided (we will explain this in the MVVM pattern usage description), thus making the presentation layer a very slightly open layer.
* The driver layer: the entry point in the application, also contains a stylesheet for the used controls so that a uniform design is applied over the app. Because we used a DI container, we configured it globally, giving our application a tree – like structure of dependency injection, making any component accessible (except the data models) through a simple injection in the class constructor.

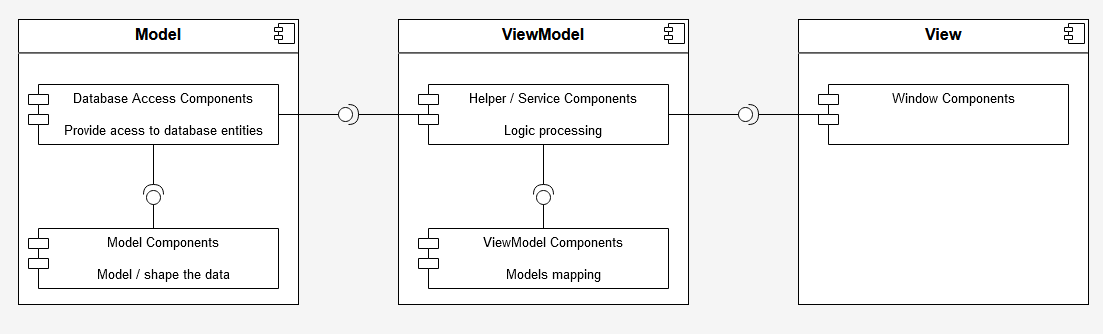
Because we developed a desktop application using the WPF framework, it felt naturally to use the MVVM pattern for making full use of this framework. However, there was a tradeoff between the ease of development produced by the use of the MVVM pattern together with the layered architecture: because we need certain data to be passed between windows (the logged in user, the tournament that is selected by the manager in the manager UI), we access the data layer at the transitions between windows. For this matter, the business layer is slightly open. We could avoid this by re-querying the database at each window transition (with the use of the helper services from the business layer) or by adding an additional layer of abstraction. But, due to limited time for application delivery and performance considerations, we chose not to do this and allow the business layer to be open in a very small amount.

* The Model component is mapped over the data layer from the layered architecture.
* The View component is mapped over the presentation layer of the layered architecture.
* The ViewModel component is mapped over the business layer. The ViewModels represent the mapping of the data that is displayed to the user in classes format.

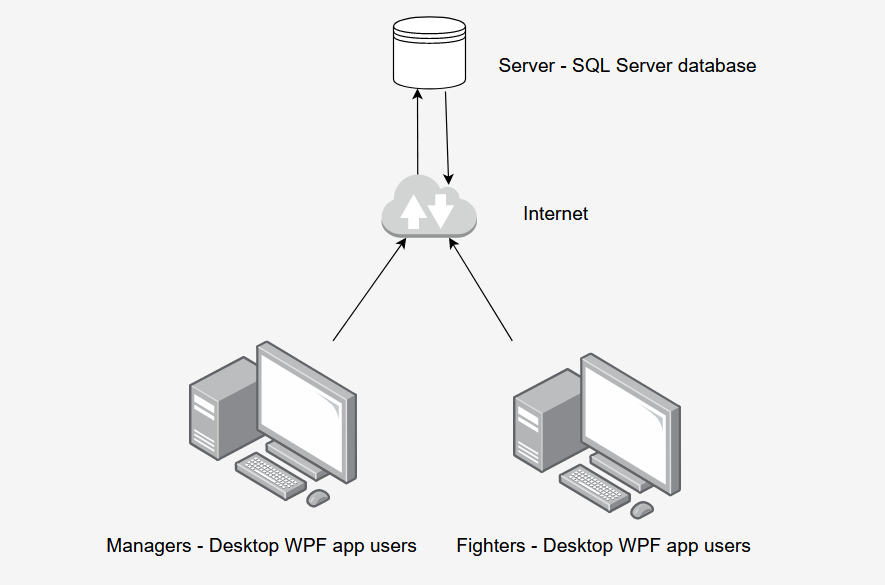
**3.2 Diagrams**

**Package diagram:**

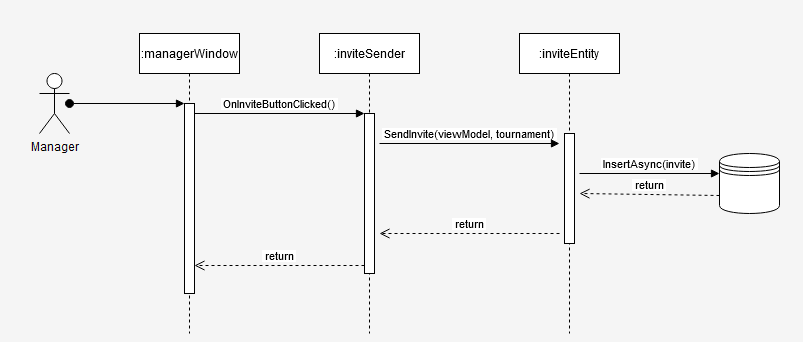
**Component diagram:**

****

**Deployment diagram:**

****

4. UML Sequence Diagrams



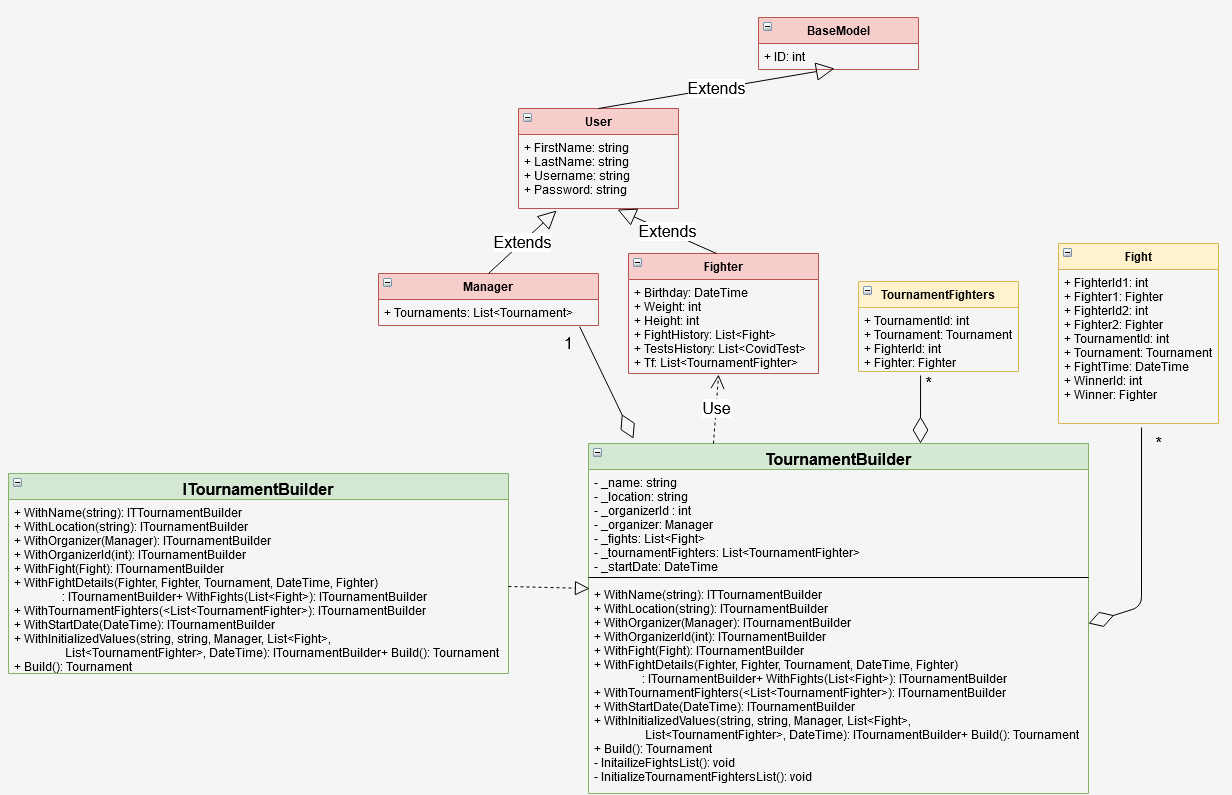
5. Class Design

**5.1 Design Patterns Description**

Because the tournament entity represents a quite huge aggregate, we need a creational pattern for managing its instances. For this purpose, we chose to use the builder design pattern. The Tournament entity consists of an Id (auto-assigned by the database, so it won’t be included here), the name of the tournament, its location, a foreign key / reference to the manager that organizes it (and because we are using EF Core we need to provide both an integer ID and a reference to a Manager object), a list of fights (one to many relationship in the database), the starting date and, because we have a many – to – many relationship between tournaments and fighters (a fighter can take part in multiple tournaments and a tournament has more fighters registered to it), we needed an intermediate entity which will be mapped to a list of TournamentFighter entities (see diagram from chapter 6).

Because we are using a DI container at the higher level of the application, the TournamentBuilder class implements the ITournamentBuilder interface, for easier dependency injections all over the application. The diagram for the pattern is displayed below (chapter 5.2).

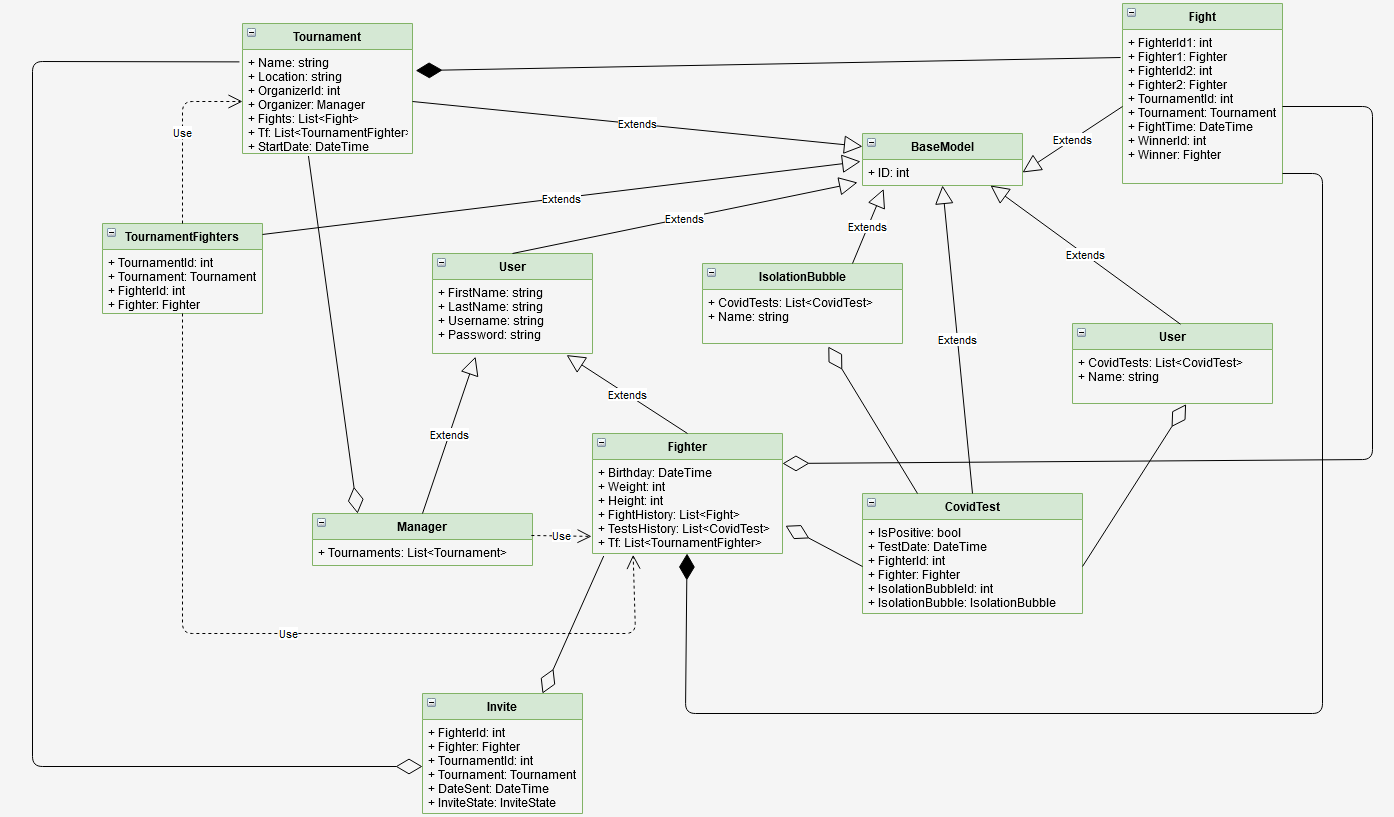
On this diagram the TournamentBuilder and the implemented interface are colored in green. On the same diagram the Player-Role design pattern will be explained (or rather the variation used in the application).

**5.2 UML Class Diagram**

The Builder design pattern provides a higher level class that maps its content (fields and methods) around the built object. The fields are the properties of the tournament that can be set using the builder, while the methods are basically setters for the inner representations of those fields in the builder class. However, each such setter returns an instance of ITournamentBuilder, so that the successive calls of these setters can be chained (such as the methods that perform operations on streams), until a final operation is called, which retrieves the build version of the tournament instance.

On the same diagram the previously mentioned variation of the Player – Role design pattern can be noticed. Although the Role class misses, the principle is the same: we have a common ancestor: the User class, that would have a dependency on the Role class, if we implemented it, and then the sub – rooles of Manager and Fighter which a user can take. However, because the roles cannot change over time, the Role dependency would become redundant and we can use the type of the class at runtime to determine the role of a certain User instance.

6. Data Model

**

The data layer in our application, the Models package more precisely, mirrors almost perfectly the data shape we have in the database. However, some additional properties are created and not mapped in the database (the [NotMapped] annotation in EF Core). These properties are used for easier computation of certain information (if a fighter is eligible for taking part in tournaments, for instance), but they are used in the application, not from the data’s point of view. In the diagram above we removed these properties and left only the data properties that reflect the database entities and their relationships (foreign keys, dependencies etc.).

7. System Testing

For testing we proposed a small number of tests, due to time limitations on the project delivery. However, the critical parts of the project were tested: the database connection, the builder design pattern functionality and the eligibility of a fighter for taking part in tournaments. The unit testing approach was unit testing, using NUnit testing.

* Database connection: we established a database connection and we checked against a select command (retrieve an instance we know is in the database from the setup) and later delete it. A cleanup method was provided to ensure no testing entities remain in the database.
* For testing the builder design pattern, we created manually a instance of a tournament, and then another one with exactly the same data, but using the builder. Finally, a comparison between them should return true.
* For testing the eligibility of a fighter, we test 3 cases:

1. The fighter is eligible (has no positive test and has tested regularly over the last 3 weeks).
2. The fighter is not eligible because a test was found positive.
3. The fighter is not eligible because he was not tested regularly over the last 3 weeks (at least one test each week).

Future releases of the application will contain unit testing for larger areas of the project, including the services (the helpers from the business layer), other database accessors etc.

8. Bibliography

1. **Using Autofac demo**: <https://www.youtube.com/watch?v=mCUNrRtVVWY>

2. **Using certain EF Core features**: <https://docs.microsoft.com/en-us/ef/>