Assignment 3

Analysis and Design Document

Student: Bidian George Alexandru

**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 3

3. System Architectural Design 3

4. UML Sequence Diagrams 3

5. Class Design 3

6. Data Model 3

7. System Testing 3

8. Bibliography 3

1. Requirements Analysis

# Assignment Specification

The aim of this project was to design and implement an application that helps MMA tournaments manage their scheduled fights better while ensuring covid safety standards.

A manager should be able to create a tournament and invite fighters to sign up for a venue. Each tournament requires weekly or monthly 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 (weekly or monthly).

# Functional Requirements

The web application is welcoming the user by asking them to choose which type of user they are (fighter/administrator). This process is used instead of a login system. After picking their category, in case the user is a fighter, they will get to a simple form page. On this the user is required to introduce information about themselves like: first name, last name, weight and the date of the last negative covid test taken. The field requires a date, but the system is only trusting the fighter’s honesty. After all the data is introduced, the fighter clicks a register button and is successfully registered and inserted in the data base.

When entering the application as an administrator, the user will be welcomed by a page containing the following buttons: Tournaments, Fighters, Covid Tests and Schedule fighters!. The first 3 buttons mentioned previously access different pages which allows the admin to visualize, delete or update the raw data from the data base.

The last button will lead the user to the main functionality of the application: the match making part. On this web page are 4 input fields: tournament title, tournament location, tournament date and tournament type(which is a radio buttons group). After introducing the necessary information, the admin will be able to submit the data by clicking the Schedule button which will create a match(fight) between 2 fighters, at a period specified by the type selected: monthly/weekly.

The process of scheduling a fighter is a bit more complex. When the admin is clicking the schedule button, the system receives all the fighters from the data base and tests them. By testing them, I mean that each fighter will receive a second covid test. This test is generated by assigning a random True or False value, with the probability of 1/10 for a True test. In case the test is true, the fighter will be moved in a quarantine bubble. While being quarantined, the fighter can not participate in any kind of event. In order for a fighter to leave the quarantined state, the generateCovidResult function will have to be negative 3 weeks in a row.

On the right of the page, there can be seen a table of fighters. This table is implemented with the help of Decorator Design Pattern. By using a DTO type object (FighterDTO), a new field is assigned to this type of object called color. If the fighter is in quarantine, then the color assigned to this fighter will be red (‘lightpink’ for design purposes), green(lightgreen) in case the fighter is not in quarantine and finally, at the beginning, the fighters are colored in a gray shade, meaning that they were not tested yet.

After all the tests have been done, the application will start to match fighters. In order for 2 fighters to be matched, there are some conditions to be acquired: Their weight difference has to be less or equal with 5, so they have to be in the same weight category. Another important condition, which gives more safety to the tournament, is that the fighter is not quarantined.

When scheduling matches, the system will try to put 1 fight per week/month depending on the tournament type selected, by using the Strategy pattern. In case there are no fighters capable to fight the application will show an alert telling the admin that no fighter could be found for that week.

When pressing the Schedule button, the system is trying to create a match for the current date which is shown below this button. The current date is changed based on the type of tournament selected. In case it is weekly, a week will be added to the previous current week, otherwise a month(4 weeks) will be added. If the tournament just started, then each fighter will be initially tested, which means that a random test will be assigned to each fighter, in case it is true, the fighter will be moved in quarantine, otherwise the fighter can be matched with another fighter of the same category. In the case that the tournament type is monthly, the test phase will be applied 4 times when pressing the Schedule button. This proves that the process is applied during live time.

# Non-functional Requirements

CQRS can be very handy for some scenarios, and those are some benefits of CQRS:

* With the separation of concerns, helps to minimize and manage the complexity, making the application more maintainable, extensible, and flexible.
* Segregating the responsibility between commands and queries can help to improve performance, scalability, and security.
* The workloads between read and write operation will be different and using CQRS allows you to scale each of them independently of the others.
* Security is improved because you will have a single object model to execute an update operation ensuring that only the right classes will do it.
* Because each class is responsible for reads or writes, this reduces the chance of exposing data that should not available to a particular user.

**Disadvantages of CQRS:**

Implementing CQRS also have some disadvantages that must be taken into account:

* Increase the complexity of the code.
* If using messaging to process commands and publish update events, the application must handle message failures or duplicate messages.
* If you separate the read and write databases, the read database may be stale, and it can be difficult to detect when a user has issued a request based on stale read data. Even if you use event-sourcing or some other mechanism to keep the databases sync, there will be some time delay (even if it’s a small one) before the writing database be consistent. So you should consider that maybe you can be reading data that is stale.

Extensibility is a measure of the ability to extend a system and the level of effort required to implement the extension. Extensions can be through the addition of new functionality or through modification of existing functionality. The principle provides for enhancements without impairing existing system functions. This represents a problem in our case, the project being structured on a layered architecture. Due to this architecture, it would take a serious amount of time to go through all the layers in order to extend a feature. The same principle is applied to Maintainability, these two requirements being very similar.

Deployment requirements describe the precise, desired configuration of a software system. They relate the system’s non functional requirements to its architecture, providing a basis for making decisions about design trade-offs in terms of the resulting system's non functional properties. In my case, the project also uses a Client-Server architecture, making it easy to deploy .

Another consideration with the layered architecture pattern is that it tends to lend itself toward monolithic applications, even if you split the presentation layer and business layers into separate deployable units. While this may not be a concern for some applications, it does pose some potential issues in terms of deployment, general robustness and reliability, performance, and scalability.

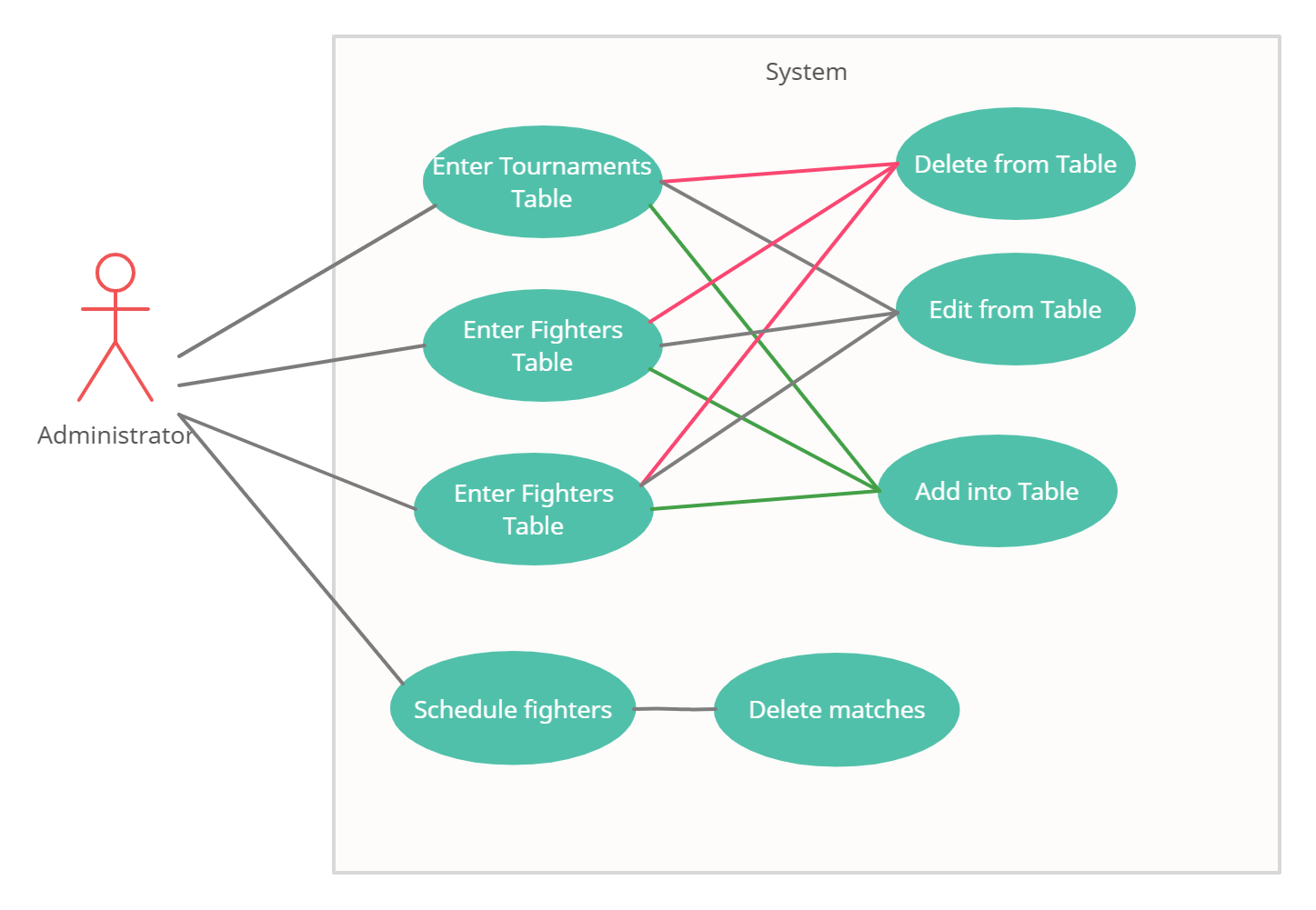
Testability is a great attribute of the layered architecture. Most of the logic is situated in the same place, so testing is much easier. Also some part of these tests are being applied in the frontend part, the visualization layer checks its input fields.

When it comes to Usability, the user is welcomed by a very friendly and basic User Interface. Everything is labeled or there are instructions to help understand what is the purpose of an input field or a button.

2. Use-Case Model

Diagram, schematic

Description automatically generated



**Use case : Administrator Use Case**

**Level : summary level**

**Primary actor : Administrator**

Main success scenario: The Administrator main goal is to create tournaments and mainly matches for the tournament. The admin will fill 3 input fields and select one radio button: one about the tournament’s title, one about the tournament’s location and one about the starting date of the tournament, plus to select one option from the group of radio buttons, referring to the type of tournament: weekly or monthly. After introducing the required data, the admin will press the Schedule button. By pressing this button, the admin creates a match. The admin is able to create as many matches as the nr of fighters that are able to fight. When there are no more fighters available, or capable to fight for the current week, the admin is announced via an alert and the Schedule button is disabled.

Extensions: When creating a match for the same tournament, the admin does not modify the inputs given previously. If these inputs are modified, the match that will be created will be assigned to another tournament, just created with the new values from the modified inputs.

3. System Architectural Design

**3.1 Architectural Pattern Description**

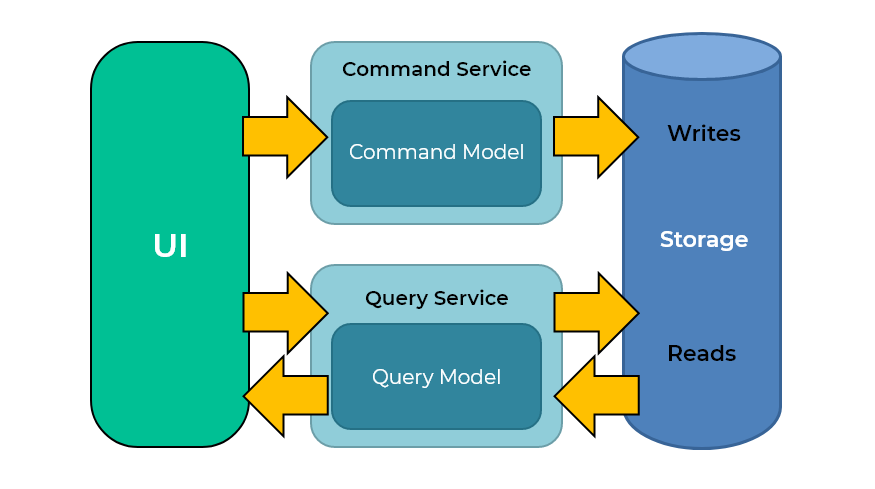
**3.1.1 CQRS Architecture**

The Command and Query Responsibility Segregation (CQRS) it’s an architectural pattern where the main focus is to separate the way of reading and writing data. This pattern uses two separate models:

* **Queries**— Which are responsible for reading data
* **Commands**— Which are responsible for update data

*The Command and Query Responsibility Segregation (CQRS) pattern separates read and update operations for a data store. Implementing CQRS in your application can maximize its performance, scalability, and security.*

The image below illustrates a basic implementation of the CQRS Pattern:



**Commands**

Commands represent the intention of changing the state of an entity. They execute operations like Insert, Update, Delete. Commands objects alter state and do not return data.

Commands represent a business operation and are always in the imperative tense, because they are always telling the application server to do something. A command is an object with a name of an operation and the data required to perform the operation. For example:

public class **DeactivateProductCommand** : ICommand  
{  
 public readonly int InventoryItemId  
 public redonly strnig Comment; public DeactivateProductCommand(int id, string comment)  
 {  
 IventoryItemId = id;  
 Comment = comment;  
 }  
}

The commands are interpreted by the CommandHandlers and they return an event, which can be a successful event or a failure event. If the command succeed it will create a success event, and if the command fails it will create a failure event.

**Queries**

Queries are used to get data from the database. Queries objects only return data and do not make any changes.

Queries will only contain the methods for getting data. They are used to read the data in the database to return the DTOs to the client, which will be displayed in the user interface.

Queries usually start with the word Get, because queries ask for the application to provide some data, for example:

public class **GetProductByIdQuery**() : IQuery  
{  
 public int Id { get; set; } public GetProductByIdQuery(int id)  
 {  
 Id = id;  
 }  
}

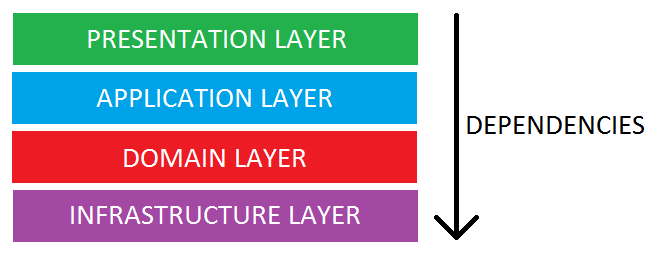
**3.1.2 Layered Architecture**

A layered Architecture, as I understand it, is the organization of the project structure into four main categories: **presentation, application, domain, and infrastructure**. Each of the layers contains objects related to the particular concern it represents.

* **The presentation layer** contains all of the classes responsible for presenting the UI to the end-user or sending the response back to the client (in case we’re operating deep in the back-end).
* **The application layer** contains all the logic that is required by the application to meet its functional requirements and, at the same time, is not a part of the domain rules. In most systems that I've worked with, the application layer consisted of services orchestrating the domain objects to fulfill a use case scenario.
* **The domain layer**represents the underlying domain, mostly consisting of domain entities and, in some cases, services. Business rules, like invariants and algorithms, should all stay in this layer.
* **The infrastructure layer (also known as the persistence layer)**contains all the classes responsible for doing the technical stuff, like persisting the data in the database, like DAOs, repositories, or whatever else you’re using.

There are two important rules for a classical Layered Architecture to be correctly implemented:

1. All the dependencies go in one direction, from presentation to infrastructure. (Well, handling persistence and domain are a bit tricky because the infrastructure layer often saves domain objects directly, so it actually knows about the classes in the domain)
2. No logic related to one layer’s concern should be placed in another layer. For instance, no domain logic or database queries should be done in the UI.



**3.1.3. Client-Server Architecture**

**Client–server model** is a [distributed application](https://en.wikipedia.org/wiki/Distributed_application) structure that partitions tasks or workloads between the providers of a resource or service, called [servers](https://en.wikipedia.org/wiki/Server_(computing)), and service requesters, called [clients](https://en.wikipedia.org/wiki/Client_(computing)).[[1]](https://en.wikipedia.org/wiki/Client%E2%80%93server_model#cite_note-1) Often clients and servers communicate over a [computer network](https://en.wikipedia.org/wiki/Computer_network) on separate hardware, but both client and server may reside in the same system. A server [host](https://en.wikipedia.org/wiki/Host_(network)) runs one or more server programs, which share their resources with clients. A client usually does not share any of its resources, but it requests content or service from a server. Clients, therefore, initiate communication sessions with servers, which await incoming requests. Examples of computer applications that use the client–server model are [email](https://en.wikipedia.org/wiki/Email), network printing, and the [World Wide Web](https://en.wikipedia.org/wiki/World_Wide_Web).

The "client–server" characteristic describes the relationship of cooperating programs in an application. The server component provides a function or service to one or many clients, which initiate requests for such services. Servers are classified by the services they provide. For example, a [web server](https://en.wikipedia.org/wiki/Web_server) serves [web pages](https://en.wikipedia.org/wiki/Web_page) and a [file server](https://en.wikipedia.org/wiki/File_server) serves [computer files](https://en.wikipedia.org/wiki/Computer_file). A [shared resource](https://en.wikipedia.org/wiki/Shared_resource) may be any of the server computer's software and electronic components, from [programs](https://en.wikipedia.org/wiki/Computer_program) and [data](https://en.wikipedia.org/wiki/Data_(computing)) to [processors](https://en.wikipedia.org/wiki/Microprocessor) and [storage devices](https://en.wikipedia.org/wiki/Data_storage_device). The sharing of resources of a server constitutes a *service*.

Whether a computer is a client, a server, or both, is determined by the nature of the application that requires the service functions. For example, a single computer can run web server and file server software at the same time to serve different data to clients making different kinds of requests. Client software can also communicate with server software within the same computer.[[2]](https://en.wikipedia.org/wiki/Client%E2%80%93server_model#cite_note-2) Communication between servers, such as to synchronize data, is sometimes called [*inter-server*](https://en.wikipedia.org/wiki/Inter-server) or *server-to-server* communication.

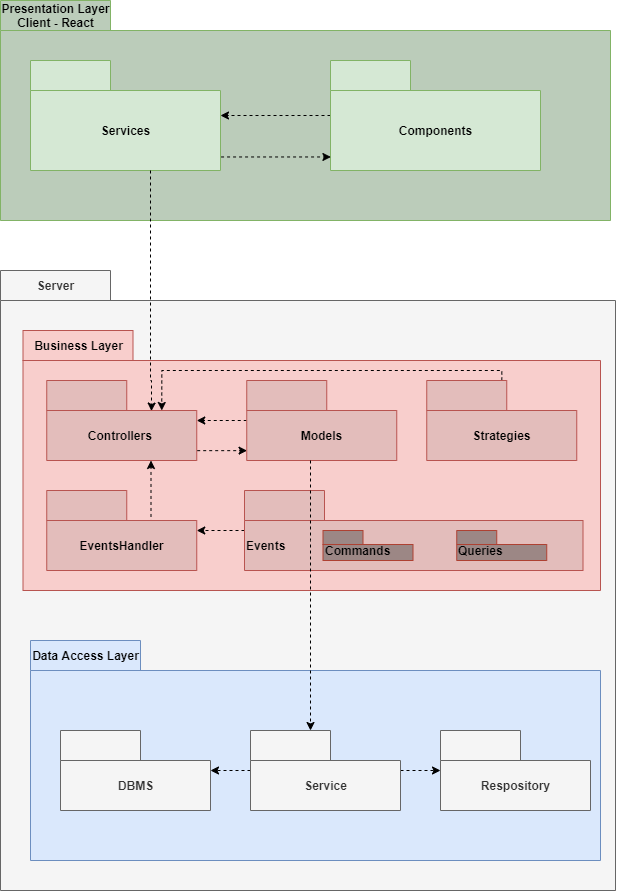
Graphical user interface, application

Description automatically generated

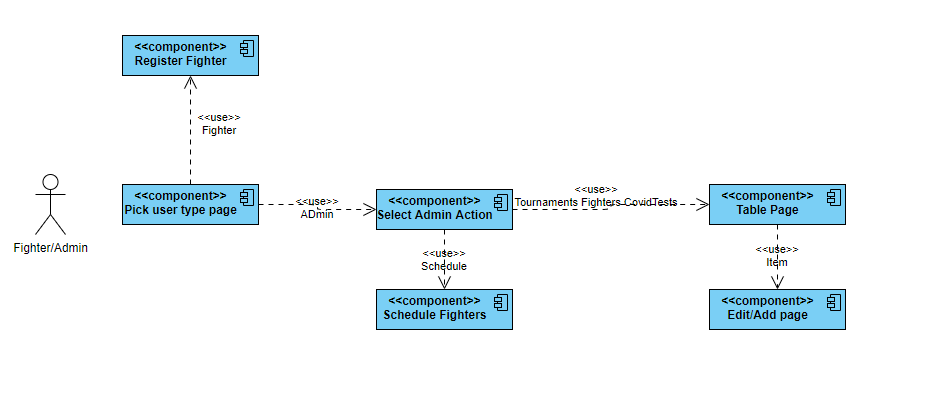
**3.2 Diagrams**

The structure of three structural architectural designs can be seen in the project. The layers of the architecture can be observed throughout the application. The presentation layer contains the frontend which is done in React and represents the Client part of the Client-Server architecture, while the other packages represent the Server part. The business layer contains the controller of the application, strategy classes and its logic classes, the model (Fighter, Tournament, CovidTest etc.) The Data Access Layer contains the interfaces that link the database and the application, mostly using CRUD Repository and for the Covid Test the JPA Repository. The access is given through Service type classes for each model class. Finally there is the Database Layer, containing, as the name suggests, the DB used for the application.

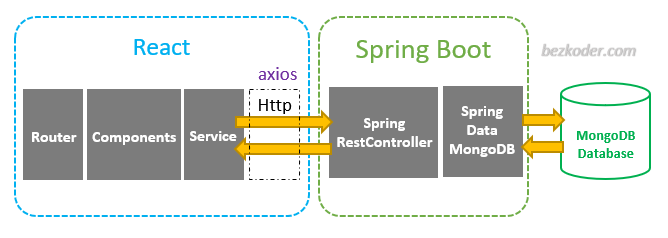
**3.2.1 Package Diagram**

****

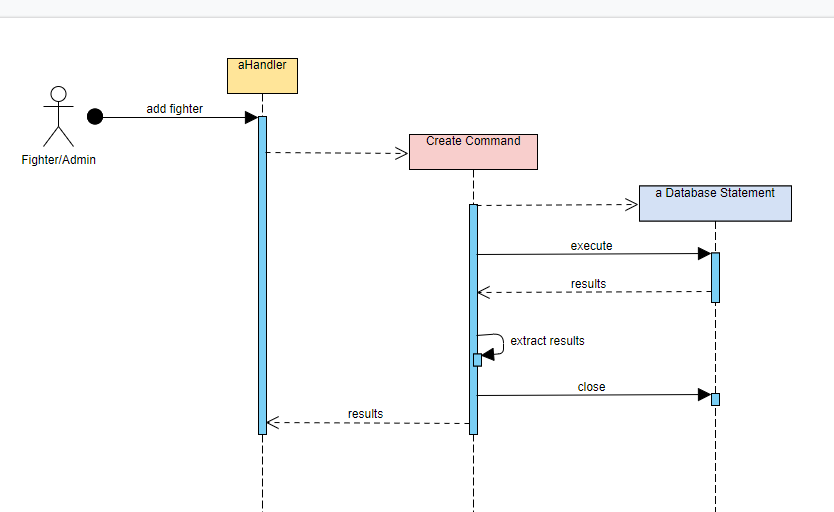
**3.2.2. Component Diagram**

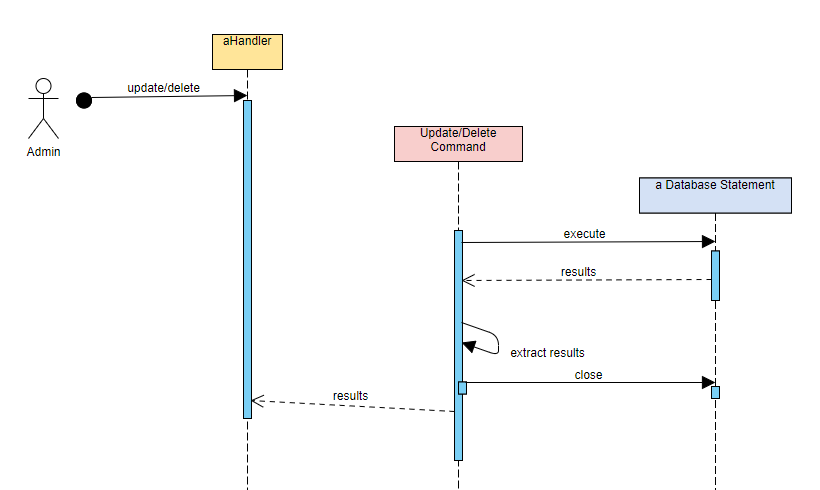
****

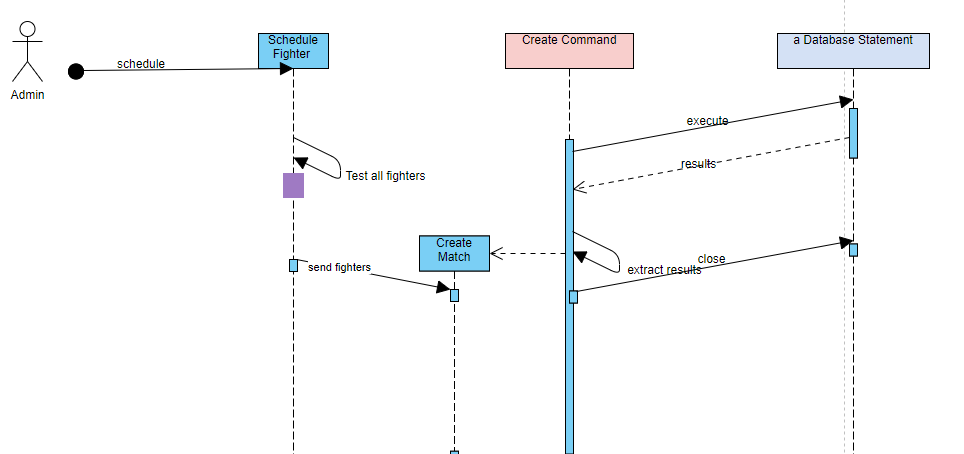
**3.2.3. Deployment Diagram**

****

4. UML Sequence Diagrams







5. Class Design

**5.1 Design Patterns Description**

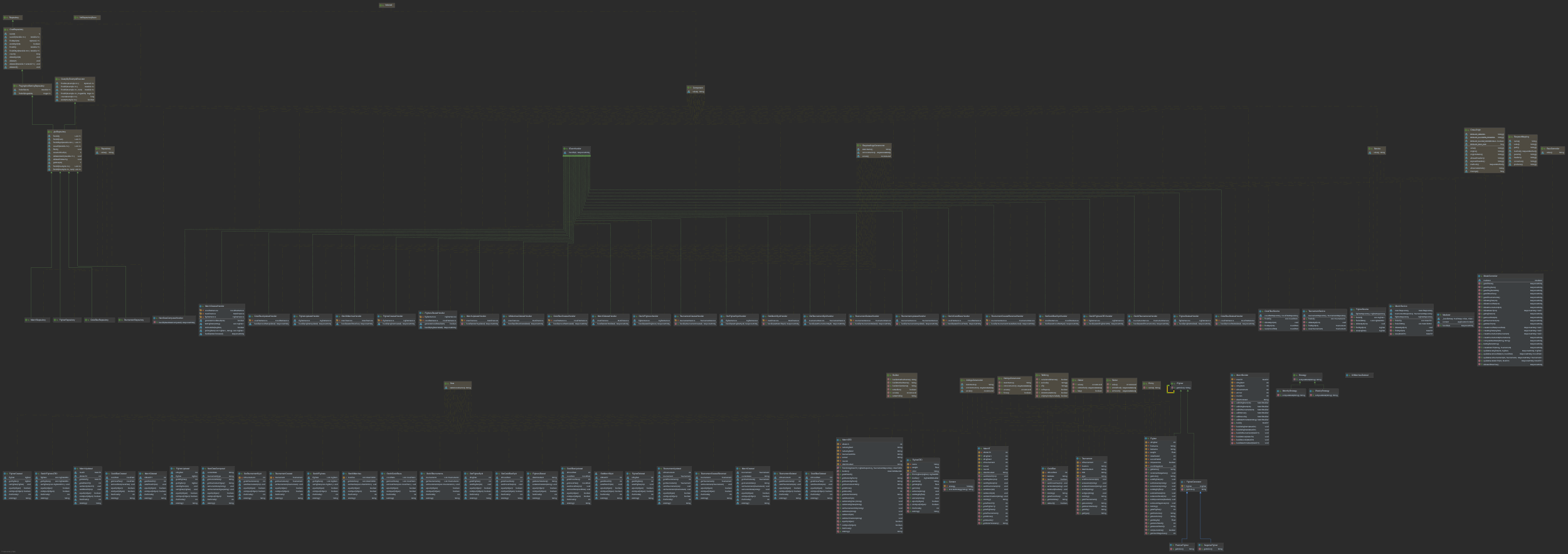
A design pattern I used in this project is the Builder Design Pattern.The builder pattern is a design pattern designed to provide a flexible solution to various object creation problems in object-oriented programming. The intent of the Builder design pattern is to separate the construction of a complex object from its representation. It is one of the Gang of Four design patterns.

Another design pattern used is the Strategy Pattern. The **strategy pattern** (also known as the **policy pattern**) is a [behavioral](https://en.wikipedia.org/wiki/Behavioral_design_pattern) [software design pattern](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)) that enables selecting an [algorithm](https://en.wikipedia.org/wiki/Algorithm) at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.

Another design pattern used is the DTO pattern design. The **Data Transfer Object Design Pattern** is one of the enterprise application architecture patterns that calls for the use of objects that aggregate and encapsulate data for transfer. A Data Transfer Object is, essentially, like a data structure. It should not contain any business logic but should contain serialization and deserialization mechanisms.

Another design pattern used is the Decorator pattern design. **Decorator** is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors. Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class. This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

**5.2 UML Class Diagram**

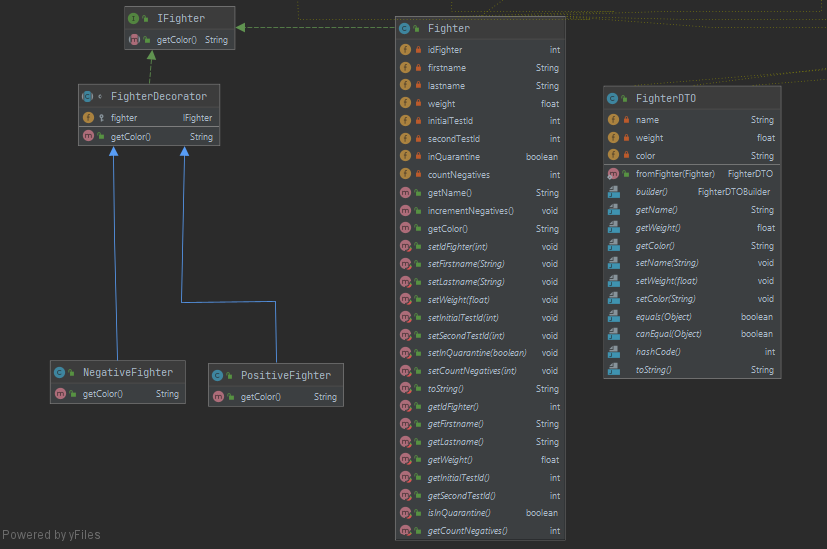


Click on the following link to see the whole diagram: <https://svgshare.com/s/XKK>.

The Builder Design pattern is used to create an object of type MatchT. Such an object would have a very complex constructor, but using the pattern the process is easier. Above that, some fields are automatically generated by the builder. Such an example could be the field winner, which is randomly assigned by the builder, taking a random value between idFighter1 and idFighter2. Another value constructed in the builder is the number of rounds, which is a random value between 1 and 5. The Builder Design Pattern is also used for the MatchDTO, but this time using the Lombok @Builder annotation.

The Strategy Design Pattern is used in the Controller, having the role of converting a String which represents a date into a future date based on the requirements, either after a week or after a month. The date is taken as a String, parsed in the methods to a LocalDate and then based on the chosen strategy, the method addWeeks(1) or addMonths(1) is applied to the LocalDate. After this addition is done, the date will be formatted to a String of pattern “yyyy-MM-dd”.

The DTO Design Pattern is used in order to get a more presentable Match object. The mainly reason for using this design pattern is to display a match in such a way that it makes sense for a non-administrator user: instead of showing the tournament id of a match, show the title, and based on the same logic, show the fighter names instead of their ids. The same process is applied when using the FighterDTO. We use it to transfer a fighter object containing only the fullname, which is the fighter first name + last name, the weight and the color assigned to the fighter by using the Decorator Design Pattern.



Extending a class is the first thing that comes to mind when you need to alter an object’s behavior. However, inheritance has several serious caveats that you need to be aware of.

* Inheritance is static. You can’t alter the behavior of an existing object at runtime. You can only replace the whole object with another one that’s created from a different subclass.
* Subclasses can have just one parent class. In most languages, inheritance doesn’t let a class inherit behaviors of multiple classes at the same time.

One of the ways to overcome these caveats is by using *Aggregation* or *Composition*  instead of *Inheritance*. Both of the alternatives work almost the same way: one object *has a* reference to another and delegates it some work, whereas with inheritance, the object itself *is* able to do that work, inheriting the behavior from its superclass.

In this case, the Fighter Decorator class extends the NegativeFighter and PositiveFighter classes. The Fighter class implements the IFighter interface, representing the basic class of the decorator pattern. It has the role to create the default value for the getColor() method, which is the key method of this pattern, in this project. The Fighter class will get a gray color, the NegativeFighter will get a green color and the PositiveFighter a red color. Instead of assigning these colors at the compile time, the Decorator Design Pattern helps us assign them at runtime.

6. Data Model

The model is made out of 4 classes: Fighter, CovidTest, Tournament, MatchT, plus the MatchDTO and the FighterDTO. Each of this class has assigned to itself the @Entity annotation, which means that the name of the field will represent the name of the column from the database.

The fields of each class is the following:

* Fighter
  + idFighter – int - primary key
  + firstname – String
  + lastname – String
  + weight – float
  + wins – int - (I did not succeed implementing a functionality regarding this field)
  + initialTestId – int – foreign key – the ID of the first CovidTest assigned to the fighter
  + secondTestId – int – foreign key – the ID of the second CovidTest assigned to the fighter
  + inQuarantine – boolean – value which tells if the fighter is isolated in a quarantine bubble
  + countNegatives – int – value of the consecutive negative covid tests
* CovidTest
  + idCovidTest – int – primary key
  + testDate – String – date format is required, this value is parsed into a LocalDate
  + result – boolean – tells if the covid test is positive or negative
* Tournament
  + idTournament – int – primary key
  + title – Stirng – not null
  + location – String
  + dateTimeStart – String - date format is required, this value is parsed into a LocalDate
  + type – String – can only be “Monthly” or “Weekly”
* MatchT
  + idMatchT – int – primary key
  + idTournament – int - foreign key – ID of the corresponding tournament
  + idFighter1 – int – foreign key – ID of the first fighter assigned to this match
  + idFighter2 – int – foreign key – ID of the second fighter assigned to this match
  + winner – int – foreign key – the ID of the winner of the fight, equals idFighter1 or idFighter2
  + dateTimeStart – String - date format is required, this value is parsed into a LocalDate
  + rounds – int – value of maximum 5
* MatchDTO
  + idMatch – int
  + nameFighter1 – String
  + nameFighter2 – String
  + tournamentTitle – String
  + winner – String – name of the winner – equals either nameFighter1 or nameFighter2
  + rounds – int – value of maximum 5
  + dateTimeStart – String - date format is required, this value is parsed into a LocalDate
* FighterDTO
  + name – String – the first name and the last name of the fighter
  + weight – float
  + color – String – value that can be either lightgray/lightgreen/lightpink, value generated by the decorator

7. System Testing

The first set of tests was done during the process of creation. In order to verify if the HTTP requests work correctly, I used Postman to send requests. The Postman app was used at this time in order to assure the correctness of the code and because there was no frontend yet.

Data tests have been introduce in many parts of the backend, for example before a match is created the fighter is tested for multiple reasons: checks if the fighter has been scheduled already, checks if the second fighter is in the same weight category, and most importantly if the second test of the fighter is before the date of the match.

Multiple input tests are done in the frontend part. All of the input fields will sign the user that there is an error if the input is empty. In case of the administrator, when editing tables or adding new items into the tables, some regex verifications are made. For example: names can not have numbers in the input fields, also weight or id’s values can not have letters. Boolean fields allow only “true” or “false” values, etc.

8. Bibliography

<https://docs.google.com/document/d/171D9FdPZ_h0vI1w4l_bVOYVTKxDMkuozLIfU1w-SXVk/edit>

<https://en.wikipedia.org/wiki/Builder_pattern>

<https://dzone.com/articles/layered-architecture-is-good>

<https://www.springboottutorial.com/spring-boot-react-full-stack-crud-maven-application?fbclid=IwAR1_EEVsMkGDDM1qEVtTXEC2puVLmiWAatgQzOl1kDD6OiZeIUfLvvaydus>

<https://stackabuse.com/data-transfer-object-pattern-in-java-implementation-and-mapping/>

<https://en.wikipedia.org/wiki/Strategy_pattern>

<https://www.tutorialspoint.com/design_pattern/strategy_pattern.htm>

<https://henriquesd.medium.com/the-command-and-query-responsibility-segregation-cqrs-pattern-16cb7704c8094>

<https://refactoring.guru/design-patterns/decorator>

<https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm>