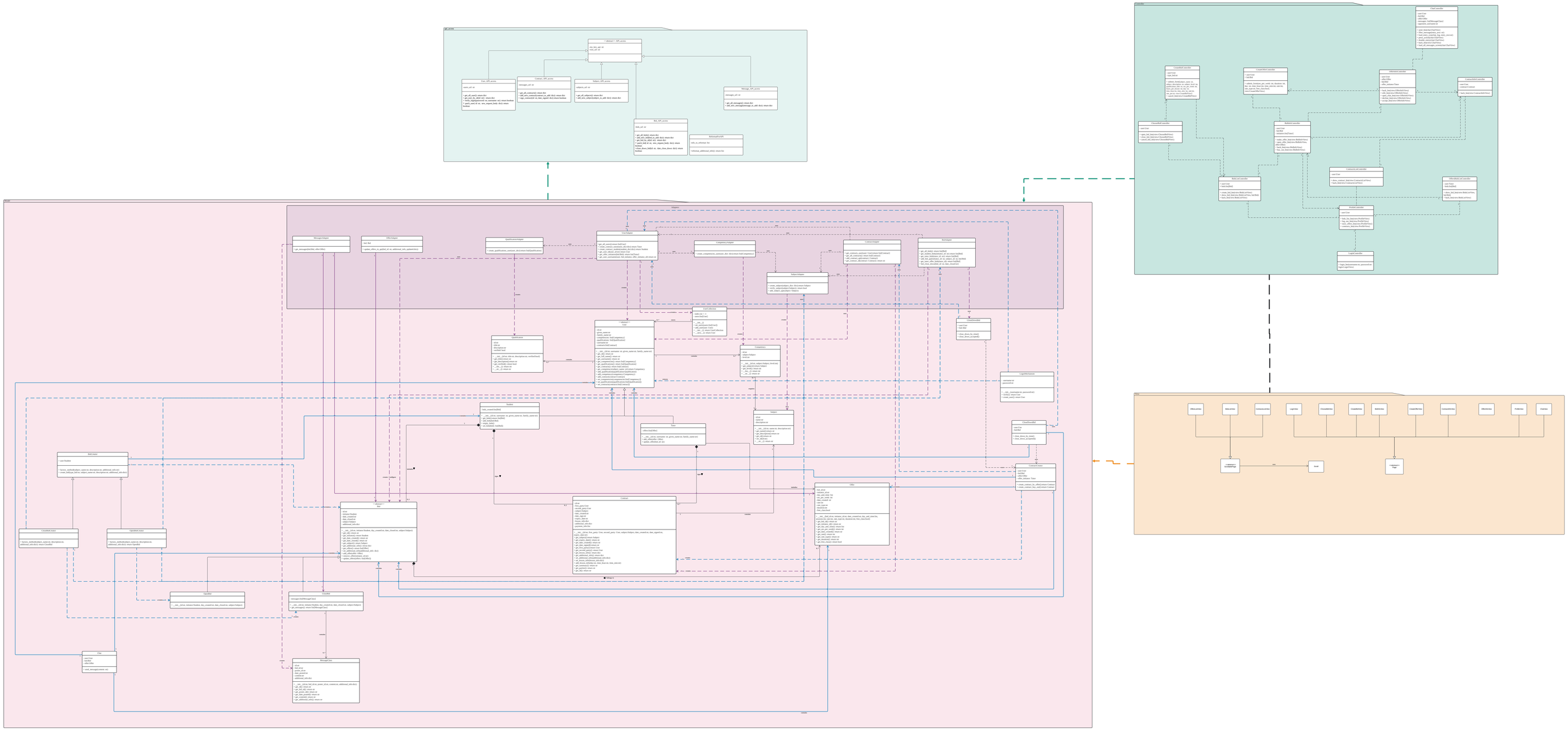
Design Rationale

Team EaT

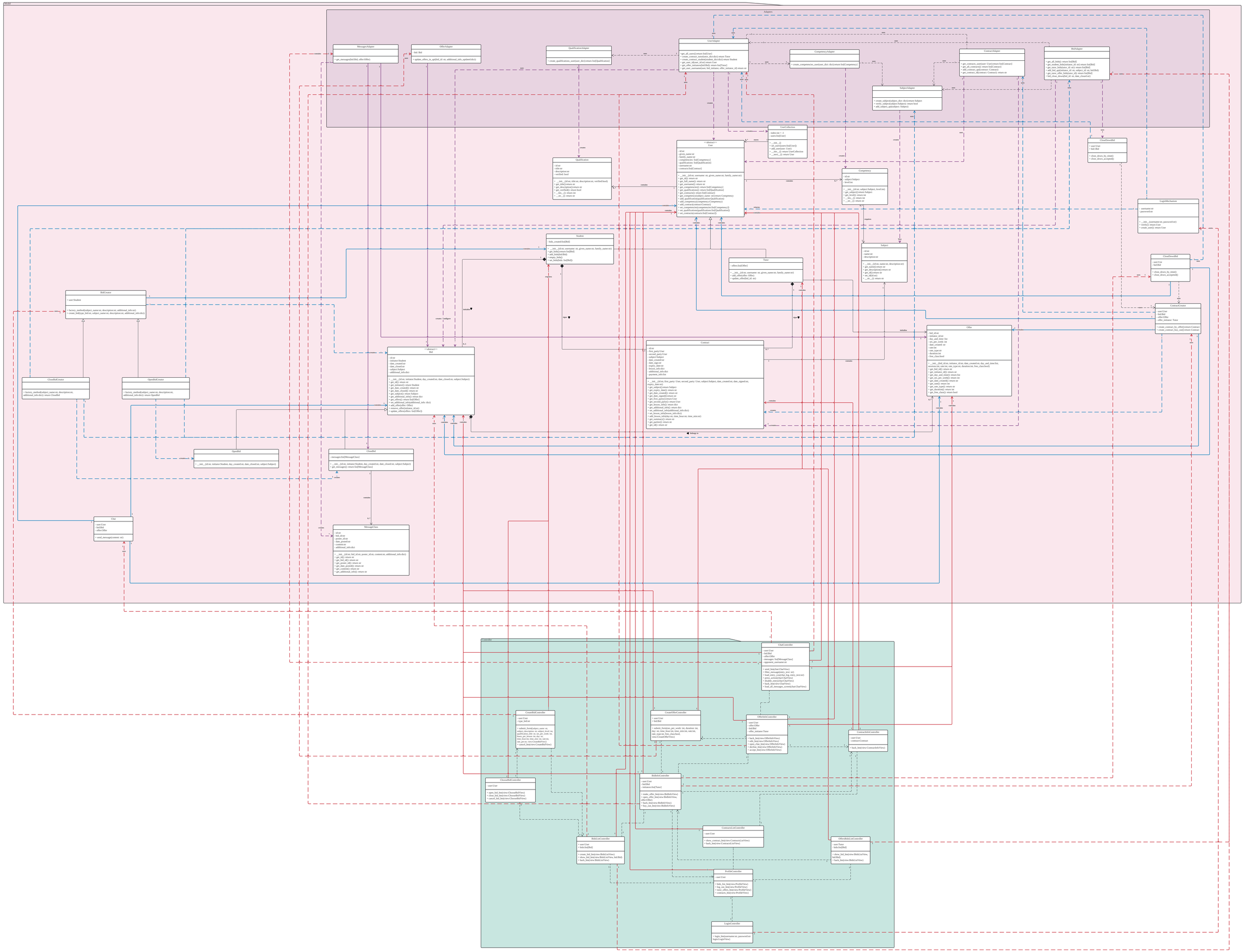
Tatiana Sutulova - 30806151

Elaf Abdullah Saleh Alhaddad - 30163977

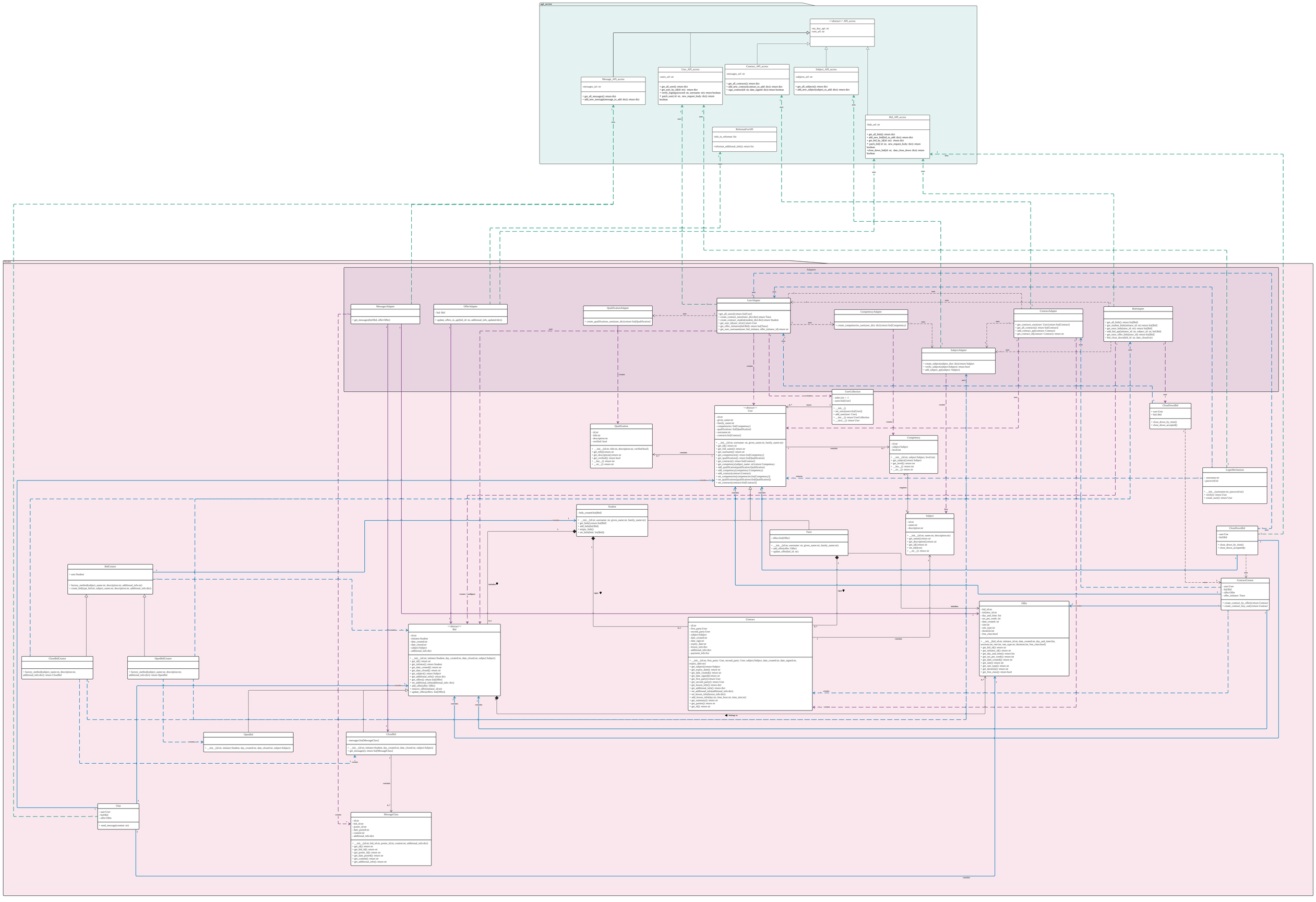
Full system Class Diagram



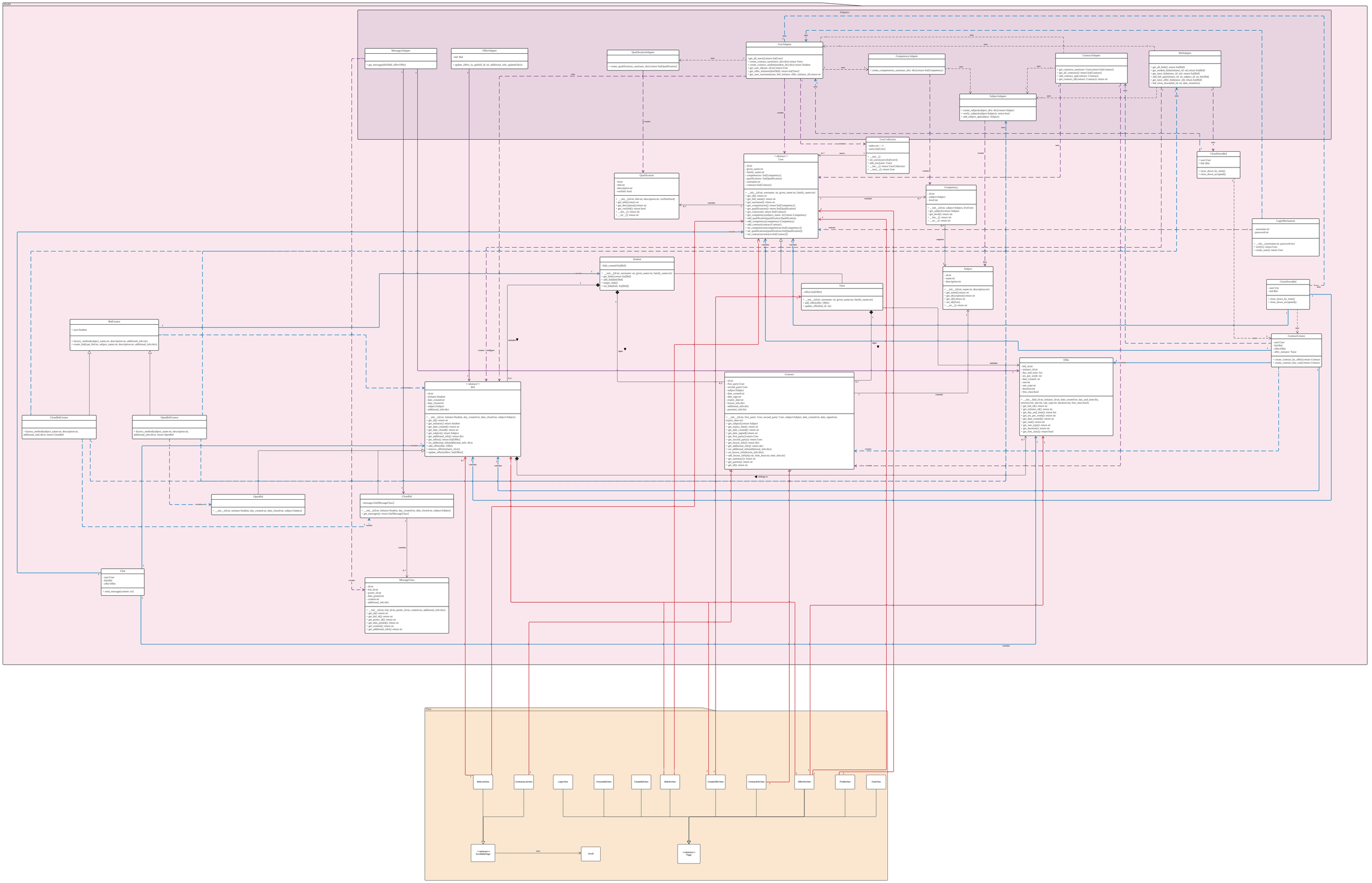
Model and Controller Diagram



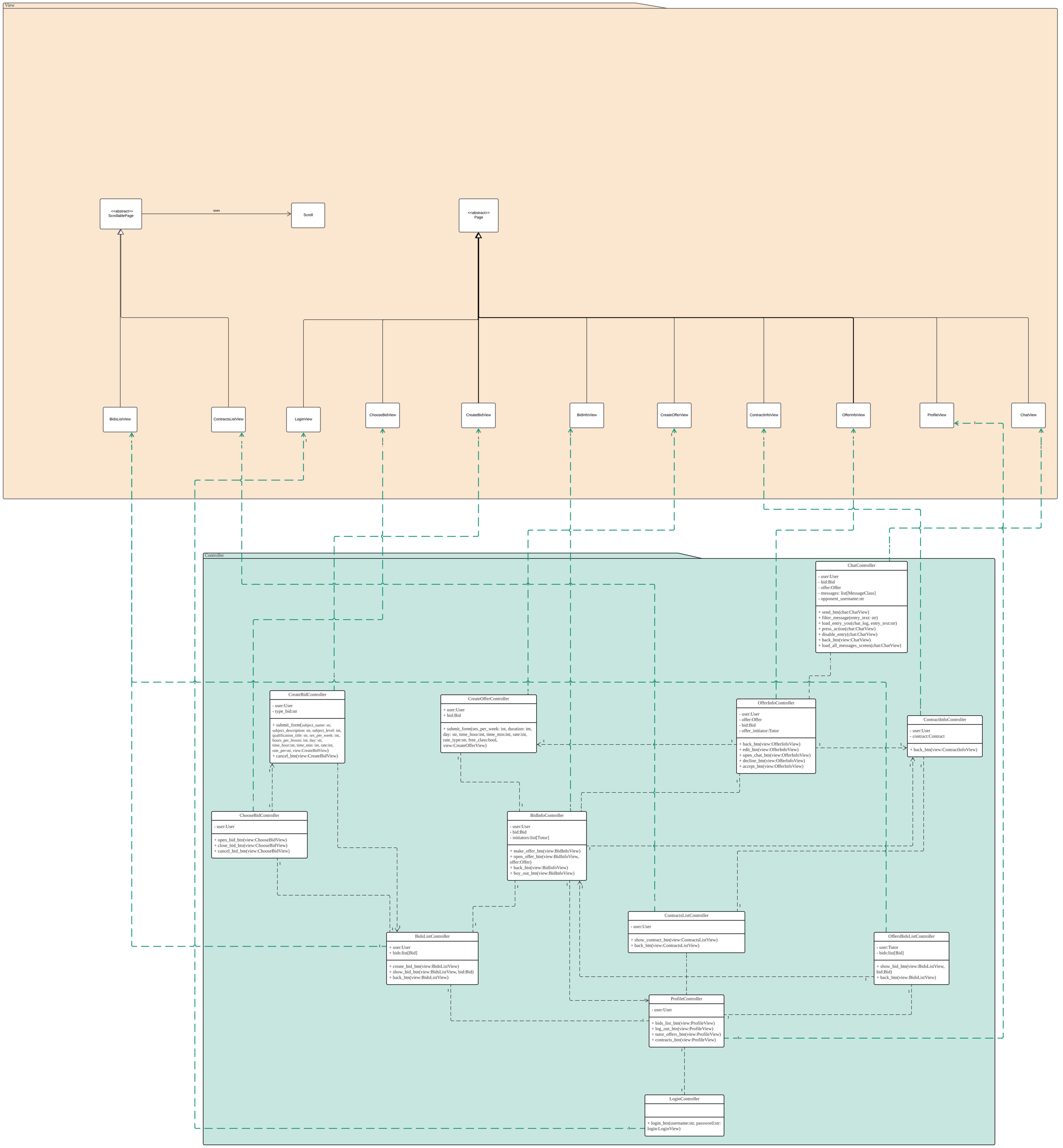
Model and API Diagram



Model and View Diagram



Controller and View Model



1. Architecture

1.1 MVC architecture

In order to build a stable and maintainable system we decided to use the MVC architecture that is made out of three packages; Model, Controller and View. In order to reduce the dependencies and frequent API call requests, we added an API package to our system architecture.

1.1.1 Model

Our Model package responsible for managing the behavior and the data of the application. It includes all the classes that will be represented in the system as well as all the classes which will be changing the state of the application. The model class also includes classes that will process the main functionalities like; Login mechanism, Opening bids, etc.

1.1.2 Controller

Our Controller package main responsibility is interpreting the user input; button clicks and keyboard inputs. The classes in the controller then inform the model classes to process the data and update the view classes accordingly. This indicates that our controller package is dependent on the model package and on the view package, since it works as the middleman between them.

1.1.3 View

Our View package acts as a user interface by representing all the data to the user and takes in the interactions that are made by the user. The package is dependent on the model and the controller. This creates a cyclic dependency between the controller classes and view classes. However, this due to the fact that we are used tkinter in creating the GUI. In tkinter the buttons in the view classes can only be controlled by the controller classes by passing the controller class to the view. The view class will also need to pass itself back to the controller class.

1.1.4 API

The API_access package is used to work with the api by containing methods that can access and update it. This package is not dependent on any other packages and is only accessed from the Model classes to be able to read and write data from/to the API when the user updates the current state of the app. The reason for that is that the data has to be updated not just locally but in the server in order for other users, who are using the other devices to see the updates made.

1.1.5 Advantages of MVC

The benefits of the MVC architecture for our application are the ease of extension and therefore ease of development. Since the UI requires more frequent changes, the model, which does not depend on it will not be affected in any way and the scope of the change is restricted to View. Moreover, using MVC made the testing of Model easier, since the Logic and the Presentation modules are separated, the UI is not needed to track the changes in the state of the application. The software architecture also allowed us to make independent parallel development of the application.

2. DESIGN PATTERNS

2.1 Adapter pattern

We used adapter classes in order to get the information in json/dictionary format and create an instance of the class with it. The main benefit of it is that it separates the data conversion code from the primary business logic of the program, ensuring that the classes are following the single responsibility principle (SRP). An example of the implementation is the classes LoginMechanism and UserAdapter. The login mechanism calls the UserAdapter to get all the users available in the system, it will then check for the presence of the user and allow the user to login into the system. This means that the UserAdapter will be responsible for configuring the Json return from the API and creating User classes while the loginMechanism will verify the presence of the account and allows the user to login, giving both classes one responsibility.

Furthermore, Adapter Pattern allows the implementation of the Open/Close principle implementation, since the new types of adapters may be integrated into the program when there is a change made to the API without breaking the logic of model classes. For instance, if the server used for this software is to be changed and the code would have to be converted to another format rather than json, the existing code would not need any changes, since new adapter classes can be added.

2.2 Factory method

Factory method pattern was implemented in the bid creation mechanism; there is a BidCreator class, which has the abstract factory method and create_bid method. There are two concrete creators: OpenBidCreator and CloseBidCreator, which are inheriting the factory method and they create concrete Bid objects. When adding a new bid, BidCreator class's create_bid method is called in order to check what type of the bid is requested and the concrete bid creator class is called from there accordingly. The following pattern makes the program easily extendable, since if the new type of bid is added, there will be no changes to the existing classes and in order to implement it, we will just have to add a new bid class and new bid class

creator. This follows the Open/closed principle(OCP) making the system open for extension but closed for modification. It also follows the single responsibility principle (SRP).

3. Design Principles

3.1 SRP

Most of the classes in our system are following the SRP, since they have only one responsibility and only one reason to change. For instance, all the controller classes have only one task, which is to manage and control the view. This includes reading buttons clicks and calling the mechanisms accordingly, collecting the input data and passing it to creator classes and passing the updated data to the view if the state of the application is changed.

The View classes have no other responsibility but representing the data to the user. As well as, a class that represents each component of the system added to the model package and other classes that will be responsible for implementing the functionalities of the system.

3.2 OCP

OCP principle states that the system should be open for extension and close for modification, meaning that the system should allow to add new features and support new functionalities, however it should be done in a way that does not change the existing code. As it was mentioned previously, one of the examples of how the following principle is applied is the bid creation system. Another example is the User and it's types: Student and Tutor. Since the user is an abstract parent class and Student and Tutor are its children classes that are inheriting from it and adding extra functionalities for themselves, which means that in the case the system is extended to require a new type of user we can simply add a new child class of the User class without making any changes to the existing classes.

4. Violation of cyclic dependency

Other than our controller classes, some Model classes in our design violate the cyclic dependency principle. For instance; a User class contains a list of contracts making it dependent on the Contract class and a Contract class contains the parties involved in signing the contract making it dependent on the User Class. This is because calling the API system multiple times to find the user by the user Id or the contract by the contract id will greatly slow down the system and will create a

difficult experience for the users of the system. The same reasoning applies to all other classes in the model package which includes Bid and Student, etc.