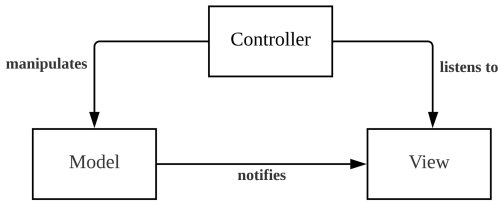
# Design Rationale

### Model View Controller (MVC) Architecture

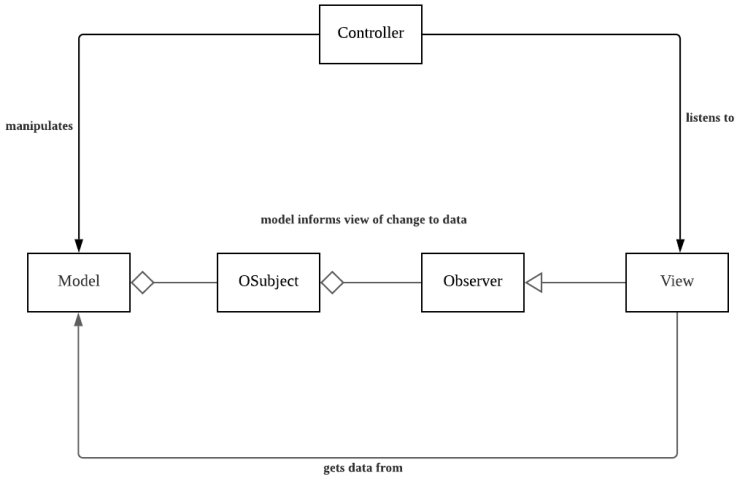



The diagram above illustrates the basic design architecture of our system. We decided to use this architecture because it facilitates **separation of concerns** between each component of the system and introduces **low coupling** between the business logic and the user input logic. Each component also has their own well-defined responsibilities (where Model contains the data, view displays the UI and controller manipulates the data). Similar reusable components are also grouped together into a package, adhering to **RREP principle**, increasing the reusability of classes.

### Interaction between classes

However, one drawback of this pattern is that it adds more complexity to the design when more and more dependencies are added, making the package/components/classes to be bloated and might violate **Single Responsibility Principle**.

To reduce such dependencies, we defined strict protocols for communication between each module.




<u>Controller and View</u>
The View informs the controller of user actions through *EventListeners.* Therefore, the View does not need to be aware of the concrete implementation of the controllers.

<u>Controller and Model</u>
The controller does not interact with any data logic, it merely acts to validate and transfer data from the view to the model. For each user action, there is one corresponding function in the model that are called that completely encapsulates the logic.

Model and View

Because the View has to have an instance of the Model to obtain data to present to the user, we decided to implement the observer pattern to facilitate the model informing the view of data change. That way, the model is not dependent on the concrete implementation of the view.

**Observer Pattern**

Observer pattern is used to ensure that the UI (views) always displays the latest information according to the model. When the Student refreshes the views, the Bidding models are updated, such as in the scenario when a Bid is closed (eg. buy-out by tutor), it notifies the views that are attached to this model and automatically updates the user interface. This advantage of this pattern is that it creates a layer of abstraction as the Subject and Observer objects do not need to know the concrete classes that observe it. One disadvantage in our existing approach is that there is only one model that observes one view, thus does not project the concept of observer pattern fully. However, such design is said to be extensible because we could easily add new types of observers that observe the views without changing much of the existing implementation.

**Adapter Pattern**

Adapter Pattern is used in the (internal) process of serialization and deserialization where JSON strings are deserialized into objects to facilitate the process of development. It helps to bridge the incompatible types (of JSON and objects), combining two independent interfaces. This provides a layer of abstraction where JSON strings are automatically converted into objects to be used in the application, and ensures loose coupling. [1]

**Singleton**

Singleton is used in the ApiService as a method of abstracting out the access to the API endpoints, creating a single access point of the API. This is similar to the **Broker Pattern** where the ApiService acts as a middleman between the server and the application. The advantage of using Singleton is that it prevents the need to instantiate multiple Api access points (UserApi, BidApi, ContractApi, etc.), where it becomes difficult to manage over time.

**Open-Closed Principle**

The BiddingModel provides an interface for new form of bidding types in the future (eg., semi-open bidding). As such, this class is open for extension where new Bidding type can be extended easily by providing its own implementation logic and functionality, but the BiddingModel is closed for modification. This creates a layer of abstraction which ensures loose coupling between classes. Similarly, other implementations such as the BasicAPI also allows new forms of API to be added easily without breaking the existing code, creating ease of simplicity in the development process.

[1] A Design-Pattern Quick Reference. Retrieved from https://link.springer.com/content/pdf/bbm%3A978-1-4302-0725-2%2F1.pdf