Justification for Candidate Architecture

Introduction

The Model-View-Controller (MVC) pattern is used as the basis of our Candidate Architecture as it enables us to:

1. Encapsulate the logic layers and display methods, thereby improving error traceability.
2. Utilize controllers to better identify control flow within the client.
3. Enables different sets of client-sided controllers (such as the LoginController and WorldController) to connect with the server using a unified API Controller, allowing standardization of API calls and improving reusability.
4. Create multiple views, fulfilling the need to develop two sets of UI for both the Teacher and Student user groups.

Quality Attributes Justification

The MVC pattern also has significant advantages for a project of this scope including:

1. Separation of view, controller and model objects, improving development speed by allowing parallel development of different components of the system.
2. Clear data-access methods available to multiple controller objects, improving scalability of functions.
3. Editing certain sections of partially developed subsystems will not affect other subsystems due to encapsulation of differing systems.

1. **Maintainability**: with distinct separation of functionalities of each class and each subsystem, updates and changes made to any class within a subsystem will not affect other classes in other subsystems.
2. **Reusability**: Reusability can be achieved by ensuring no two classes within our system overlaps in functionality but rather, taps on existing classes should there be a need for an existing method. An example of reusability in our system is the reusing of methods in the 'LeaderboardController' when a leaderboard is required. This ensures no duplicity of functionalities in other classes.
3. **Testability**: With loose coupling between subsystems, testing of individual subsystems improves, thus improving the testability of the system.
4. **Portability**: Portability can be achieved as a result of using a web server. As data is only stored on the server, the client can access the game on different devices while maintaining accuracy and consistency of their progress/data.

Other Integrated Architecture(s)

In the identification of the Candidate Architecture, we considered the following alternative architectural patterns that could be integrated:

1. Object-Oriented Model
    a. The Object-Oriented Model's concepts were integrated into our MVC model through the use of distinct objects within the Data Access Object and the various Controllers that are used to interface between subsystems.
    b. We minimized the disadvantages of the Object-Oriented Model, namely the inability to determine all classes and objects required for system execution and reuse by using the MVC model to group objects into subsystems and allowing them to use the MVC model's interfacing to improve reusability.
2. Layered System
    a. The MVC can be considered as an extension of a 3-tiered layered system with the Model, View and Controller each as a layer. The level of abstraction in the MVC model is clearly defined as each layer is further grouped into controllers and views for each respective subsystem, enabling clear distinction of the layers and subsystems.
3. Client and Server
    a. The Client-and-Server architecture is integrated into our Candidate Architecture with the MVC model adapted to include the API controller that communicates with the router in the web server, resulting in the ability of our system to handle multiple clients easily. This fulfils the requirement for the system to allow multiple clients to share data on a single server, allowing us to execute critical functions such as the Leaderboard and Discussion Board, by using a central database that is linked to our client-side MVC model using an integrated Client-and-Server architectural pattern.

Other Excluded Architecture(s)

Other architectural patterns were not included in the Candidate Architecture as they did not fulfil the requirements or did not add value to the existing system; with reference to the requirements specified in the System Requirement Specifications.

1. Pipe-and-Filter pattern
    a. The pipe-and-filter pattern is not required as there is no need for concurrent execution within the client or server. The data processed within client-server API calls is minimal, with a small amount of data processed within a relatively quick timeframe. The forced use of the pipe-and-filter pattern may also result in poor functionality due to the need to support correspondence between multiple separate but related streams.
2. Main program and subroutine
    a. The main-program-and-subroutine pattern is not preferable to the object-oriented pattern. as this project is larger in scale and does not support reusability as much as the object-oriented pattern. As the main-program-and-subroutine pattern is more complicated to implement and does not provide any significant performance

improvement over the object-oriented pattern with reference to this particular project, it is not used.

Implementation of External Frameworks

The MVC model will be implemented using the following frameworks to achieve the Candidate Architecture:

1. The Godot Framework will be used to create the View Layer. The Godot Framework allows us to encapsulate the View layer on the client-side, allowing us to effectively introduce working UI elements. Godot also allows us to implement UI elements through the Scene system, which necessitates object-oriented programming.
2. MySQL is the preferred choice of relational database management system as it allows us to effectively implement the database that will interface with the Data Access Object implemented using the Eloquent ORM.
3. The Eloquent ORM within the Laravel framework will be used to interface between the database and client requests. It is a PHP framework implementing ActiveRecord that allows us to effectively interface with the MySQL database existing on the server. It will be used to implement the Data Access Object in the data access layer on the server. It also generates the object-oriented models that will be used for timely access and modifications to the server-sided database.