**Explanation of architecture:**

The game server uses the ASP.NET Core platform (the C# equivalent of Swing for Java) and can run on Windows, Linux, and MacOS. The game client uses unity and can be build on the mentioned platforms as a desktop application, and it can also be built using WebGL to run in a web browser application served by the game server.

The server and clients communicate through web sockets. Upon receival of a web socket request, the server’s and client’s message handlers will parse the event name and data and call the appropriate event handler. The result is an event driven approach

The architecture of the game uses the Model View Controller (MVC) principle to handle the decoupling of the game logic and the Unity UI. The server game controller registers different events, manipulates the quest game and players models, and sends the required updates to the view to the Unity game client. The sole responsibility of the client is to update the view when the server sends out any events, and to notify the server when a user has interacted with the UI. In this implementation of MVC, the server’s game controller’s event handlers serve as the controller, the game logic that the game controller manipulates serves as the model, and the Unity client serves as the view.

The main advantages of the game’s architecture is that it is portable, with the server being able to run on most platforms, as well as the client being able to run on most platforms as well as web browsers. The UI and the actual game code are completely decoupled. The server side game logic and the client side UI have no awareness of each other, and the UI can be easily swapped out with any other platform supporting the web socket protocol, provided it respects the contracts for handling events sent to and received from the server.

**Class diagrams and interactions:**

The game controller class manipulates the game model (quest game class) and updates the view by sending relevant information and prompts to the game client. The game controller acts as a **mediator**, listening to messages from the game client and sending the appropriate message to the game. For example, when the controller receives a discard event from a player, it will call the relevant players discard method in the game model**.**

The following server class diagram with CRT cards explains the interactions, patterns, and abstractions used for the server side game logic.

Since the client has very minimal responsibilities, a client class diagram is not included, but the interactions for handling sending and receiving messages is similar to the server side.

**Client and server interaction:**

There is a contract in place for handling socket messages. The message body must be a UTF8 encoded JSON object with the field “event” containing an event name string and a field “data” containing any arbitrary JSON object. This contract allow for the client and server to register a dictionary of event handlers using an abstraction function .On(“event\_name”, JsonObject data) to register handlers.

On the server side each socket is mapped to a player and each player is mapped to a game as well, allowing the event handlers to know the player sending the event. This also allows events to either be broadcasted privately to a single player or to be broadcasted to an entire game.

The following client server interaction diagram explains the process of a host player creating a game, adding an AI, and having another networked player join, and running a single stage quest.

**Use cases:**

