**Discussion of Design Patterns**

We’ve used several design patterns to solve various problems. Here are the most common design patterns in our project

**Controller:**

We used the Controller pattern via a series of specialized handlers. Each handler is responsible for mediating the interaction between the UI and the domain model for a specific aspect of our application. This modular approach allows us to manage complex workflows, maintain a clean separation of concerns, and ensure our application is scalable and easy to maintain.

Handlers as Controllers: In the com.KUAlchemists.backend.handlers package, we have a dedicated handler for each operation that involves model-view communication. For example, AvatarSelectHandler is responsible for all UI actions related to avatar selection, while

PotionStorageHandler manages the interactions concerning the potion storage functionality.

**Singleton:**

We used the Singleton pattern for several components to enforce a single point of interaction. For example:

Board Class: The Board class is a perfect example of the Singleton pattern. It is designed to represent the game board, which is a shared resource among all players. There can only be one game board in a session, making Singleton an ideal choice. The Board class contains storage for ingredients, potions, and artifacts tied to the board's singular instance.

Handler Classes: Each handler class in our application, responsible for mediating between the UI and the domain model for specific operations, is also a Singleton. This design choice ensures that there is a single, consistent point of control for each operation within the game, such as selecting avatars, buying artifacts, or managing the game board.

Deck Class: Similarly, the Deck class is implemented as a Singleton because the game requires only one deck from which players draw. This prevents the inconsistencies and errors that could arise from having multiple deck instances.

**Information Expert:**

Our Board class is a prime example of an Information Expert in our system. It acts as the central authority for managing storage and state related to game elements associated with each player, such as:

Ingredient Storage: The Board maintains a mapping of players to their respective IngredientStorage, giving it the authority to query or update ingredients for any given player.

Potion Storage: Similarly, the Board is responsible for tracking each player's PotionStorage, handling the addition and retrieval of potions.

Artifact Storage: The Board also controls the mapping of players to their ArtifactStorage, overseeing the distribution and management of artifacts.

**High Cohesion:**

We used high cohesion by:

Service Classes: Each service class, like PotionBrewingService or IngredientStorageService, encapsulates all the logic related to a specific domain concept or operation. This makes our services focused and self-contained.

Model Classes as Information Experts: Classes such as Board, Deck, and Player contain data and behavior closely related to the specific concept they are modeling. They are designed to be experts in their area, thus maintaining a high level of cohesion.

Specific Handlers for Operations: Handlers like AvatarSelectHandler and BuyArtifactHandler are responsible for a single aspect of the interaction between the UI and the domain, ensuring that UI logic is not spread across multiple parts of the application.

**Low Coupling:**

We used low coupling through handlers, services, and models structural design

Our handlers communicate with the services, reducing dependencies between different parts of the system. For instance, BuyArtifactHandler will interact with BuyArtifactService without needing to know the intricate details of how artifacts are processed or stored.

The service layer acts as a boundary between our domain logic and the handlers. This abstraction allows us to change the internal workings of our services without impacting the handlers or UI components.

By segregating responsibilities into handlers, services, and models, we ensure that changes in one class or layer have minimal impact on others. For example, a change in the PotionStorage logic requires changes only in PotionStorageService and not across the entire application.

**Facade Pattern:**

We used the facade pattern idea in our code design.

Service Layer as Facades: Each service class can be seen as a facade for the specific domain it handles. For example, the forageForIngredient method in ForageForIngredientService offers a single point of interaction for the handlers or other client code to initiate the foraging action. Internally, this method deals with the Deck class to draw an ingredient and then interacts with the Board class to store the ingredient in the player's storage. All of these operations are hidden behind the simple forageForIngredient call.

The handlers interact with these service classes without needing to understand the complexities of the underlying business logic or data manipulation. This reduces the complexity faced by client code, similar to the intent behind the Facade pattern.

**Strategy Pattern:**

Our EndorseService class uses the Strategy pattern to select the appropriate naming strategy for an alchemical ingredient based on its aspects. We put strategy pattern-related classes to com.KUAlchemists.backend.strategy package.

Strategy Map: We have a map of strategies (AlchemicalNamingStrategyMap) that associates different combinations of aspects with their corresponding naming strategies as you can see from the code block below.

| **public** **class** **AlchemicalNamingStrategyMap** {  **private** **static** final Map<List<Aspect>, AlchemicalNamingStrategy> strategyMap = **new** **HashMap**<>();  **static** {  // Add other aspect combinations and their corresponding strategies  // Example: strategyMap.put(Arrays.asList(Aspect.POSITIVE\_SMALL, Aspect.NEGATIVE\_SMALL, Aspect.NEGATIVE\_BIG), new Alchemy2Strategy());  strategyMap.put(Arrays.asList(Aspect.POSITIVE\_BIG, Aspect.POSITIVE\_BIG, Aspect.POSITIVE\_BIG), **new** **Alchemy1Strategy**());  strategyMap.put(Arrays.asList(Aspect.POSITIVE\_SMALL, Aspect.NEGATIVE\_SMALL, Aspect.NEGATIVE\_BIG), **new** **Alchemy2Strategy**());  strategyMap.put(Arrays.asList(Aspect.POSITIVE\_SMALL, Aspect.POSITIVE\_BIG, Aspect.NEGATIVE\_SMALL), **new** **Alchemy3Strategy**());  strategyMap.put(Arrays.asList(Aspect.POSITIVE\_BIG, Aspect.NEGATIVE\_SMALL, Aspect.POSITIVE\_SMALL), **new** **Alchemy4Strategy**());  strategyMap.put(Arrays.asList(Aspect.NEGATIVE\_SMALL, Aspect.POSITIVE\_SMALL, Aspect.POSITIVE\_BIG), **new** **Alchemy5Strategy**());  strategyMap.put(Arrays.asList(Aspect.NEGATIVE\_BIG, Aspect.NEGATIVE\_BIG, Aspect.NEGATIVE\_BIG), **new** **Alchemy6Strategy**());  strategyMap.put(Arrays.asList(Aspect.NEGATIVE\_SMALL, Aspect.NEGATIVE\_BIG, Aspect.POSITIVE\_SMALL), **new** **Alchemy7Strategy**());  strategyMap.put(Arrays.asList(Aspect.NEGATIVE\_BIG, Aspect.POSITIVE\_SMALL, Aspect.NEGATIVE\_SMALL), **new** **Alchemy8Strategy**());  } } |
| --- |

Context Class: The EndorseService acts as the context class that uses this map to find and execute the correct strategy.

Concrete: We have concrete classes for each alchemy combination like Alchemy1Strategy, and Alchemy2Strategy. They implement AlchemicalNamingStrategy

When getAlchemicalName() is called, the service retrieves the current Ingredient's aspects and uses them to look up the corresponding AlchemicalNamingStrategy. It then delegates the call to the strategy's getAlchemicalName() method.

**Adapter Pattern:**

We’ve used the Adapter pattern to adapt the initialization of the game depending on the ApplicationMode. ApplicationMode is either ONLINE or OFFLINE. We have two types of Initializer Interfaces: OnlineInitializer and OfflineInitializer. OnlineGameInitializer implements the OnlineInitializer interface and the *onlineInitialize()* method while OfflineInitializer implements OfflineInitializer and the *offlineInitialize()* method. We have used a one-way adapter which is the OnlineInitializationAdapter class. OnlineInitializationApater class implements the OfflineInitializer interface and *offlineInitialize()* method. The constructor of OnlineInitializationAdapter takes the OnlineInitializer interface for the implementation of the adapter mechanism. We create an OnlineInitializationAdapter object in OnlineGameRoomHandler where we handle the initialization of the game for the online version. This object would be created if the user wants to play the online version of the game. Depending on the UserType(HOST/CLIENT), the OnlineInitializationAdapter creates the server or connects to the server including the initialization of the game objects. When the user wants to play the offline version of the game, *OfflineGameInitializator* handles the game initialization for the offline game.