Design Patterns

We have used the Information Expert principle because each class has responsibilities related to objects that it has knowledge of. This has led to high cohesion in our code, which is evident in die, player and board.

Our code also has low coupling, especially between the screen classes and between the game logic and the display. This means that changes can be made to the game logic without having to change the UI at all.

The methods in the Buttons class are highly cohesive since they all relate to the creation of JButton objects. Each method is responsible for creating a specific type of button, and they are organized in a way that makes it easy to add new button types without affecting the existing code. The Buttons class uses the Factory Pattern, with the getButton() method serving as a factory method that creates JButtons with different properties based on the arguments passed to it.

The createGradientPanel() method in the ColorThemes class can be considered a pure fabrication, as it does not correspond to any real-world entity or responsibility, but rather provides a convenient way to create a JPanel with a gradient background. This method encapsulates a complex process of creating a gradient background and setting it as the JPanel's background, which can be reused in different parts of the system without having to repeat the code.

The Buttons class provides a simplified interface for creating JButtons with different properties, without exposing the creation logic to the client. This makes it easier to create and manage buttons in the application and allows for easier modifications to the appearance and behavior of the buttons.

Changes could be made to accommodate more patterns. For example, the ColorThemes class currently has a switch statement in the createGradientPanel() method that chooses a specific gradient based on the name property. This could be replaced with a polymorphic approach, where each ColorThemes object has its own implementation of the getGradientBackground() method. This way, we can eliminate the switch statement and let each object decide how to create its own gradient.

In the HomeScreen class, the Creator pattern could be used to extract the creation of the buttons and their panels into a separate factory class. This would help to decouple the creation of objects from the HomeScreen class, making it easier to maintain and test.

We could have used the Singleton pattern to ensure that only one instance of a class was created. For example, we could have used this pattern to ensure that only one instance of the GameBoard class was created, and all players interacted with the same game board.

If we had more time, it would have been great to include polymorphism among the screen classes since they are all of the same kind and hence similar but have different behaviors