In our version of the implementation of the game Kivi, we built around 7 classes to handle the functionality of our game. Classes such PlaceAPiece, RollADice, PlayerSettings, and DisplaySettings were used to display the 4 different use cases we picked. All these classes enacted their utility through the usage of many GRASP design patterns we've discussed over the course of this semester.

For instance, PlayerSettings acts as the controller for player settings. It handles the responsibility of the application logic of players and their features separating it from the interface layer of the game. While managing the UI components, it also updates the bobblehead colours and ensures valid selections. This class also serves as an Information Expert as it owns difficulty dropdowns, player count, color dropdowns, giving it the responsibility to modify and validate these functionalities. This allows the game to have a much cleaner core functionality which is easier to revise and maintain in case there are changes to make later.

The class KiviGame handles the user interactions and coordination between different UI elements. It follows the Creator design pattern through its creation of various UI components such as JButton, JComboBox, etc. Additionally, it also creates the BobbleheadPanel objects within createPlayerPanels() method, which initializes related objects inside itself. As a separate, distinct class which focuses on game setup and UI management, it also manipulates high cohesion as its design pattern.

However, other design patterns that could've been better implemented are for one: polymorphism. Instead of having tightly coupled the logic around player configuration (e.g: humanCheckboxes) to the UI, we could improve the code by abstracting players into their own independent classes (HumanPlayer, ComputerPlayer, etc.) and then use polymorphism to assign each player type and its behaviour during runtime.

We could also benefit from using the design pattern 'Pure fabrication' such as in KiviGame class. Instead of the addition of multiple methods to save, load, or reset the game state directly into the class, we could introduce a StateManager class that will be solely responsible for controlling and managing the game state. This would reduce the complexity of the KiviGame class, making the system more cogent and maintainable.

All in all, the amalgamation of several design patterns that we touched upon this semester helped us curate this game up to par with the requirements for the group project. Did we strive to fulfill the necessary obligations well within the time frame? We believe, yes. Could we also further ameliorate it into perfection? Definitely. It is a continual process; art is never finished, only abandoned.