As the scale of GUI elements started becoming more grandiose, it was necessary to start breaking them into sub-components. Since libGDX has a distinct parent-child relationship between each actor, it makes sense to create DAG's that relate each component to its children. Much like the ReactJS design methodology, each component in our new dialogs can be separated logically by function and it will render itself in a certain way based on the parent it's assigned to. An example of this methodology in action exists with our PlayerHUD group. Within the PlayerHUD, there are several components like the CurrentPlayerInfo and the Scoreboard. Within the CurrentPlayerInfo it has its own PropertyTable and handles all the interaction on its own. When looking at the directional flow of each component, it always goes in one direction effectively negating any risk of having circular logic. If two components of the same group need to communicate between each other, they are required to share state at the lowest scope they have in common.

State Pattern:

Naturally, game state flows as the players progress through certain stages in the game. At the moment our state machine goes from the pre-roll phase then migrates to the post-roll phase. Eventually, there will also be an InJail phase that will function differently to prevent the player from moving on their rolls prompting alternative options. The main reason to use our state pattern is to simplify the logic within the "Roll Dice" button on the player's screen. Our roll button can simply call the doAction() method given by the RollContext and the logic to process that event lives in the TurnState methods of each child class. During the pre-roll state, the game rolls the dice and displays them using the RollDiceGroup's updateDice method. After this, the action checks to see how many times doubles were rolled during the current turn. If doubles weren't rolled, the state alters into the post-roll state but if doubles were rolled 3 times this turn, the player is sent to jail. The post-roll state servers to let the player make actions elsewhere before they end their turn such as selling or trading properties. Once the player wants to end their turn, they hit the "End Turn" button which used to be the "Roll Dice" button and it passes the turn to the next player starting at the pre-roll state.

Singleton:

Since objects spanning across the entire project need a reference to the primary state, stage, and board, we offer up a singleton class appropriately called GameState. GameState is a singleton instance which prevents having static references to objects and restricts flow between objects to expected behavior. The game state also has important methods to start the game, get the current player, and end the current players turn. Another important function of the GameState object is that it serves to update hud information when things like player money or property ownership changes. By keeping this in gamestate, there it's easy to pass this information between different classes within the project space.

Facade:

When switching between the player setup screen and the actual game screen, our code goes through a singleton named ScreenManager. The ScreenManager.showScreen method takes a Screens enum and internally facilitates the game object's screen transition.