A key design thought to our project was to work within the scope of the libGDX engine. The libGDX 2D engine uses the concept that everything functions like a theatrical play. Actors exist on Stages and perform their roles independently of each other but can reference different elements of the stage they are currently on. By working within the engine, we can avoid implementing a lot of low level rendering logic and stick in a high level context. Since we work under an engine and avoid the use of the java swing library, our rendering is highly optimized and goes through the GPU rather than rendering in memory.

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.

Factory:

Creating all the classes of the board requires a lot of special constructors as each space has its own onLand() or onClick() logic. By using a Factory pattern, we can delegate AbstractSpace creation directly from our config.json file since it contains the type information of each space. Since everything goes through the config file, it becomes extremely easy to configure and change spaces should the user want a different order of spaces. During the creation of the board, it becomes extremely easy to sequentially read through the config.json and pass those results over to our event factory; The factory will then return an AbstractSpace object which the board can position and render accordingly.

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.