**Introduction**Team 3’s project will create an executable to play checkers locally or with a bot at three levels of difficulty. The levels of difficulty will include the common trope of an “easy”, “medium”, and “hard” difficulty. If the team is able to get the project finished sooner than available we will be adding sound effects, and a soundtrack to the project.

Also David, our designer, created a [project design prototype](https://www.figma.com/proto/rUMMfdIkJUp8yJeDB0GZP8/Checkers?page-id=0%3A1&node-id=1-16&node-type=frame&viewport=-3097%2C163%2C0.6&t=Plq1M6Lycs6dazCq-1&scaling=scale-down&content-scaling=fixed&starting-point-node-id=1%3A16). It provides a rough outline visually of the final product.

**Architecture**

The checkers game application follows a Model-View-Controller (MVC) architecture, which divides the program into three main components: the Model, the View, and the Controller. This design approach enhances organization, readability, and maintainability by clearly separating the game data, logic, and user interface.

**1. Model**

The Model represents the core game data and logic. It includes the board, pieces, player information, and the rules of the game.

* Game State and Logic:
  + The Game class handles the main game state, such as tracking which screen is active, player turns, and game rules.
  + It manages the overall game logic, including initializing the board, switching turns, and checking for a winner.
* Data Storage:
  + The Board class stores the structure and layout of the game board. It maintains a two-dimensional array where each position represents a piece or an empty space.
  + The Piece class represents each individual game piece, storing information such as color and type.
* Player Information:
  + The Player class holds each player’s data, including their name and color selection. This enables a personalized experience and allows the program to manage player-specific details.

**2. View**

The View is responsible for rendering the user interface. It displays the game board, pieces, menus, and feedback to the player. By separating the View, it becomes easier to modify the appearance or layout without affecting the game logic.

* User Interface Rendering:
  + The UI class contains functions to render different screens and interface elements:
    - draw\_main\_menu(): Displays the main menu with options like "Local Play" and "Bot Play."
    - draw\_player\_setup(): Allows players to enter their names and select colors.
    - draw\_difficulty\_selection(): Displays difficulty options for gameplay.
    - draw\_board(): Renders the checkers board with pieces during gameplay.
    - draw\_winner\_screen(): Shows the winning player and allows the user to return to the main menu.
  + Each function handles the visual elements for a specific screen, keeping the rendering logic separate from the game’s core data.
* Visual Representation of Game State:
  + The View continuously updates based on the Game class state, ensuring that the display always reflects the current game situation.
* Feedback for User Actions:
  + The View includes methods to provide visual feedback for player actions, such as draw\_invalid\_move\_marker() to show an "X" when a player tries an invalid move. This improves the user experience by making the game more interactive and responsive.

**3. Controller**

The Controller is responsible for handling user input and updating the model and view based on the actions taken by the player. The controller listens to events such as mouse clicks and key presses and directs the appropriate responses within the game.

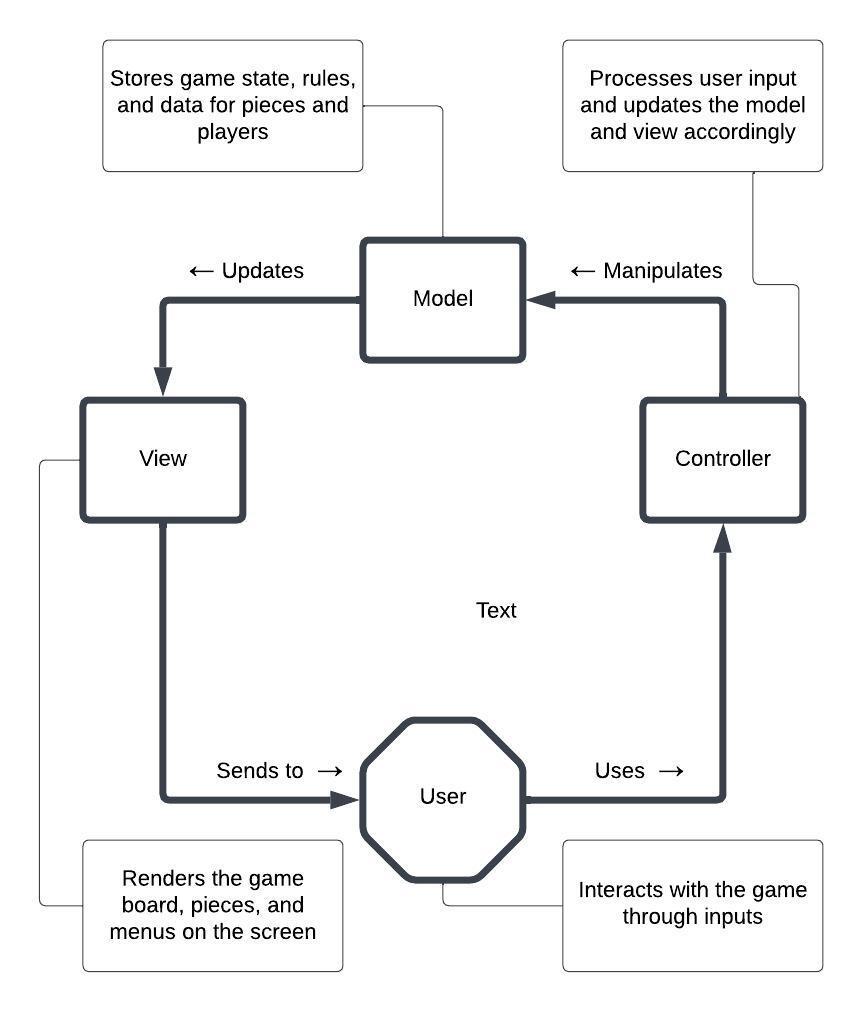
* Event Handling:
  + The main game loop serves as the controller, constantly listening for events and sending these actions to the appropriate methods within the Game or UI classes.
* Processing User Input:
  + The controller processes inputs like selecting and moving pieces, entering player names, and selecting colors. It validates actions and updates the model when a valid action is detected, ensuring the game responds appropriately.
* Screen Transitions:
  + The controller manages screen transitions based on game state. For example, it switches from the main menu to the player setup screen and then to the game board view as the player progresses through the game.

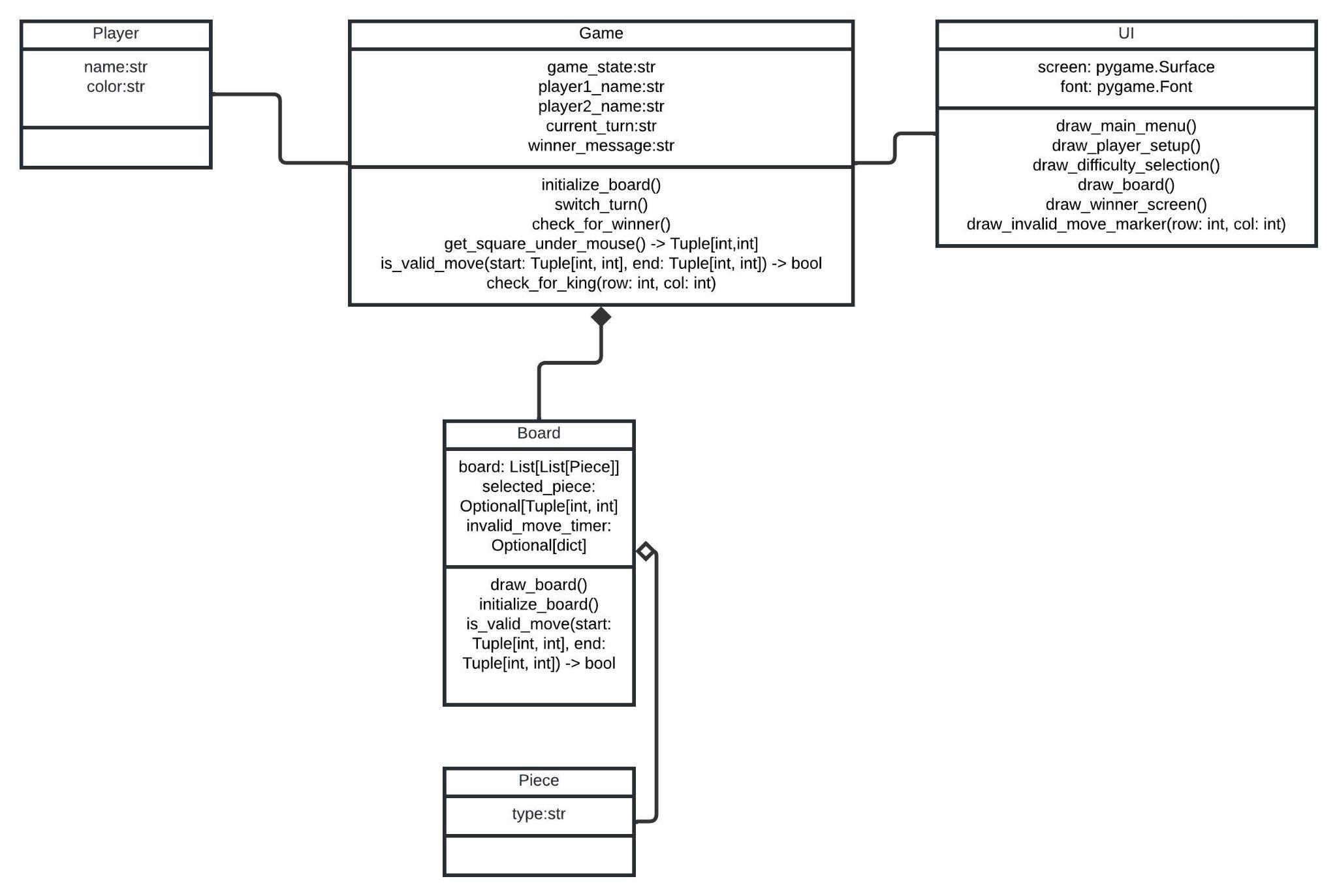
**MVC Architecture Illustration**

In this Pygame application, the MVC architecture can be visualized as follows:

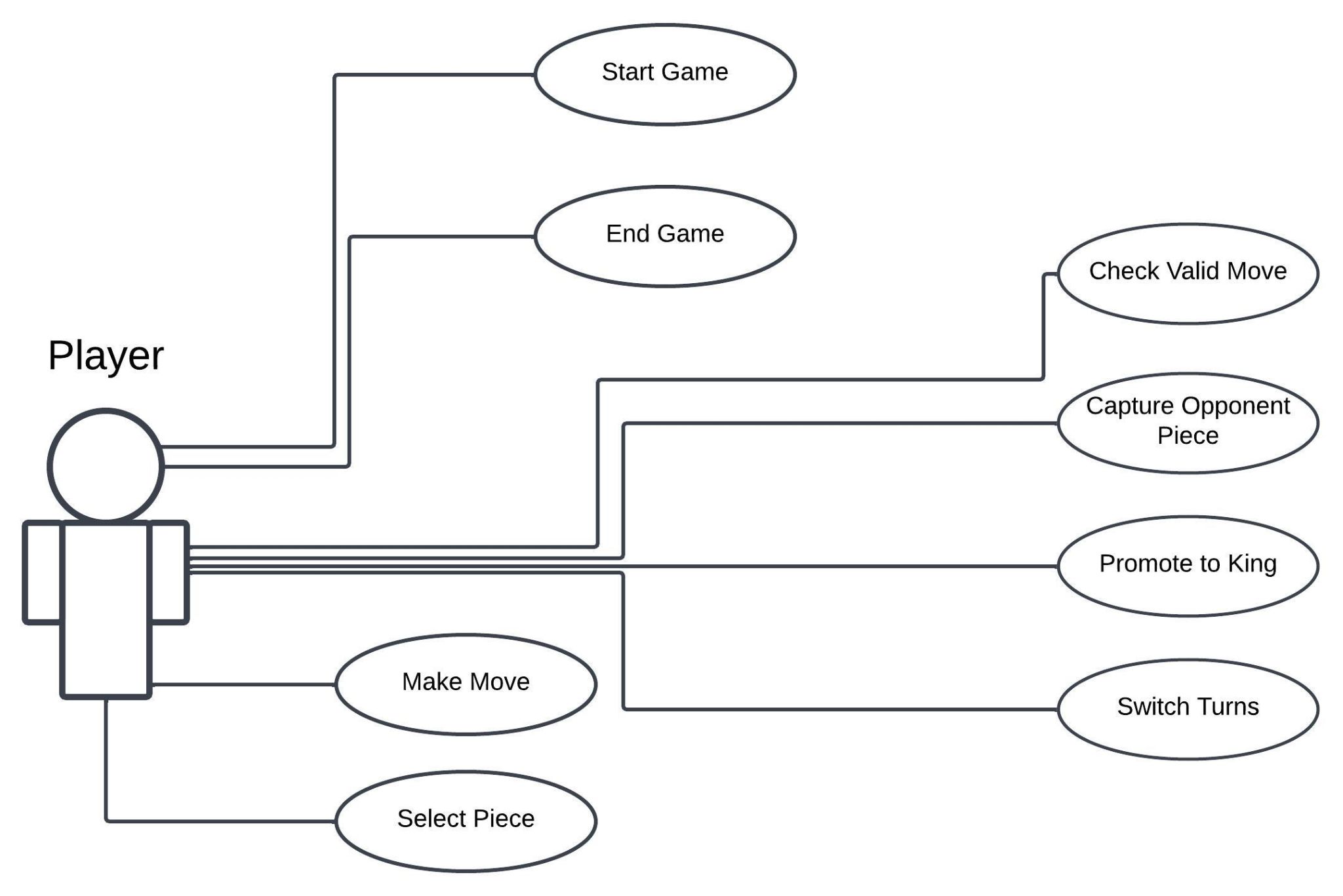
* Model: Contains classes for Game, Board, Piece, and Player to manage game data, rules, and player information.
* View: Uses the UI class to render the user interface, displaying the board, pieces, and various menus.
* Controller: The main game loop acts as the controller, handling user input and updating the model and view based on player actions.

This architecture structure allows for a clean separation of responsibilities, making the game code modular and easier to maintain or expand in the future. The MVC approach also enhances flexibility, enabling potential adjustments to the user interface or game rules without disrupting the overall program logic.



**UML**

**UML Use Case**



**Test Cases**

| Test Case # | Test Case Description | Projected Results | Direct Dependencies |
| --- | --- | --- | --- |
| 1 | Player selects piece | Player selects one of their own pieces to move. | Player has a piece to select. |
| 2 | Player moves piece | Player moves one of their own pieces once selected. | Player has a piece to move. |
| 3 | Players captures opponent’s piece | Player moves a piece to capture an Opponent’s piece. | Player has a piece to move, the Opponent’s piece is in the appropriate space to be captured by the Player. |
| 4 | Player promotes to King | Player’s piece gets promoted to King, acquiring all of the appropriate movesets. | Player’s piece reaches the other side of the board. |
| 5 | System switch turns | Once the Player’s turn is finished, it switches to the Opponent’s turn. | Player has moved a piece. |
| 6 | System checks for valid move | System evaluates if the piece the Player selects can be moved on the space the Player clicks on. | Player has a piece to move. |
| 7 | System checks for win condition | System evaluates if either Player captured all of their Opponent’s pieces. | Either Player eliminates all of the Opponent’s pieces. |

**Summary**

To summarize, Team 3 has a very accomplishable goal of a locally playable digital checkers game with the option to play with bots if wanted. Our devs have done an amazing job with defining the architecture of the project, analyzing the use cases, and structuring the UML. Our testers have also done a great job with listing the test cases so far in development. To conclude, our dev team is about to fully leave the Waterfall portion of the Waterfall-Agile hybrid approach to the project, and begin to use Agile as we remove bugs and refine the project.