FIT3077 Sprint Four User Stories and Architecture and Design Rationales Report

3 Man’s Mischief (Team 38)

Josiah Schuller, Caleb Ooi, Steven Pham

1. Revised User Stories for Advanced Requirement..... 2
Advanced Requirement..... 2
User Interface..... 3
Gameplay..... 3
2. Architecture and Design Rationales..... 4
2.1. Revised Architecture..... 4
2.2. Design Rationales..... 6
2.2.1. Advanced Requirement Design..... 6
2.2.2. Architecture Revision..... 7
2.2.3. Screenshots..... 9
Menu..... 9
AI (Player 2) Deploying..... 9
AI (Player 2) Forming mill and removing pieces..... 10
AI (Player 2) Moving piece..... 10
AI (Player 2) Fly move:..... 11
Winning..... 11

1. Revised User Stories for Advanced Requirement

As a team, we have selected “c” as the advanced requirement: *A single player may play against the computer, where the computer will randomly play a move among all of the currently valid moves for the computer, or any other set of heuristics of your choice.*

Below are the user stories we deemed to fit the advanced requirement:

Advanced Requirement

- As a player with no available friends, I want to be able to play against computers, so that I can play the game without needing another person to play with
- As a player, I want to be able to see what moves the AI has done, so that I can make informed decisions.
- As a learner, I want to be able to play against progressively harder computers, so that I can get better at the game by beating each level
- As a person who thinks they are smart, I want to be able to play Nine Men’s Morris against a hard AI, so that I can prove to myself how smart I am.
- As a player, I want to be able to select the level of the AI that I play against, so that I can play against challenging opponents.
- As an indecisive person, I want to be able to randomly choose which AI to play against, so that I can have a surprise element in my gameplay.
- As a bad Nine Men’s Morris player, I want to have AI vs AI gameplay, so that I can observe and learn from their strategies.

Below are the user stories for the software from Sprint 1, note that some have had their users changed to a fill a wider variety:

User Interface

- As a player, I want to be able to see the current layout of the board, so that I know the current state of the game.
- As a 9MM board, I want to be able to see the number of pieces that have been taken, so that I know the progress of the game.
- As a spectator, I want to be able to tell whose turn it is, so that I know which player is making their move.
- As a competitor, I want to be able to see the vacant spots on the board, so that I can decide where to put my piece in order to win.
- As a 9MM board, I want to be able to see the mills that are present on the board, so that I know which pieces are protected.
- As a player, I want to be able to see which pieces I can remove from the board when I get a mill, so that I can make better decisions.
- As a player, I want to be able to navigate the different game modes of the game with a User Interface, so that I can easily start a new game or choose another mode.

Gameplay

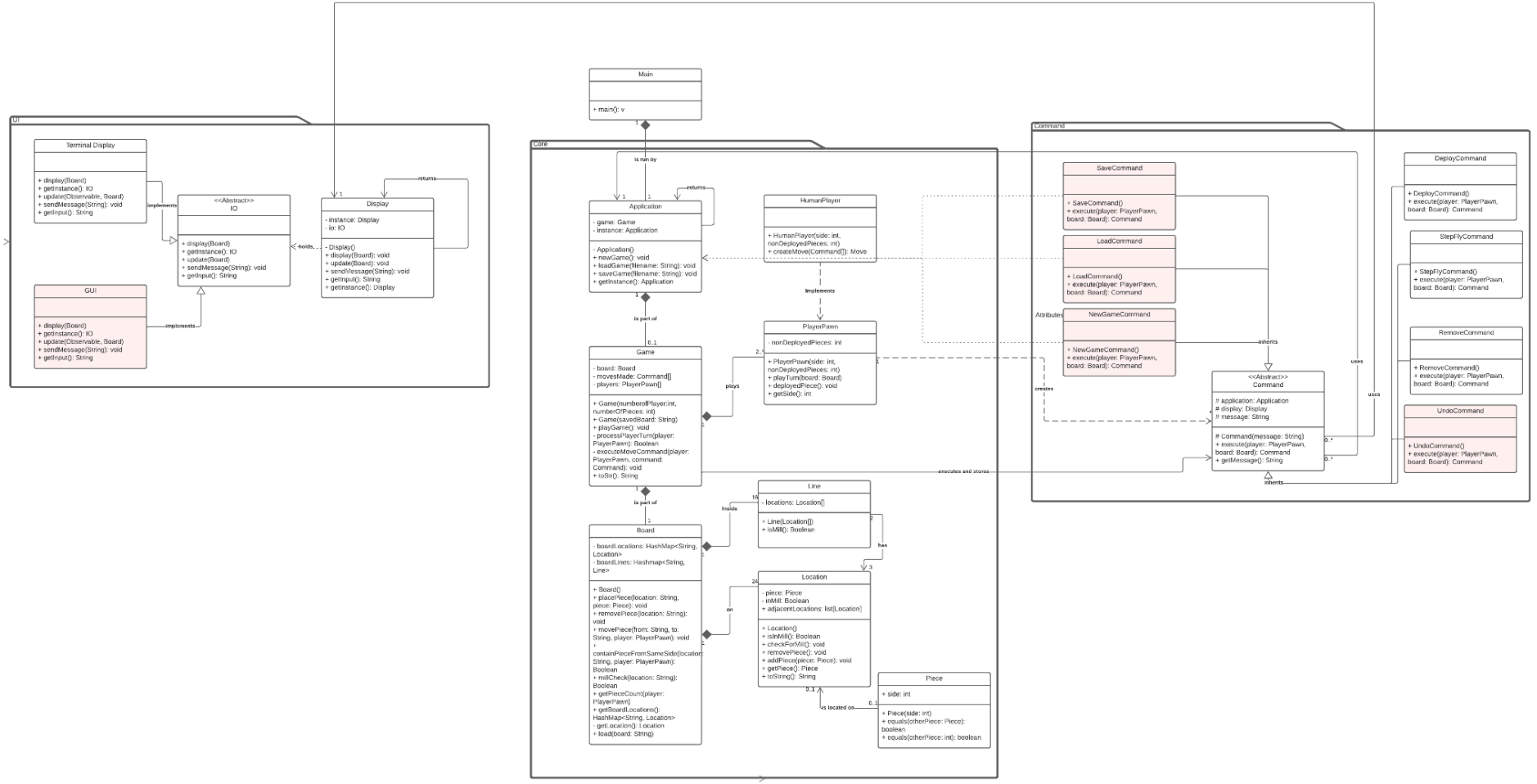
- As a 9MM piece, I want to be able to slide into vacant spots if I'm not a mill, so that I can form a mill
- As a 9MM piece, I want to be able to be placed on the board, so that moves may be played.
- As a player, I want to be able to move my piece to any part of the board when I only have three pieces left, so that I can play my moves from a weak position.
- As a player, I want to be able to remove a piece of my opponent's when I form a mill, so that I can win the game
- As a player, I want the game to end when me or my opponent has fewer than three pieces, so that a winner can be found.
- As a player, I want the game to end when me or my opponent has no more legal moves, so that a winner can be found.

2. Architecture and Design Rationales

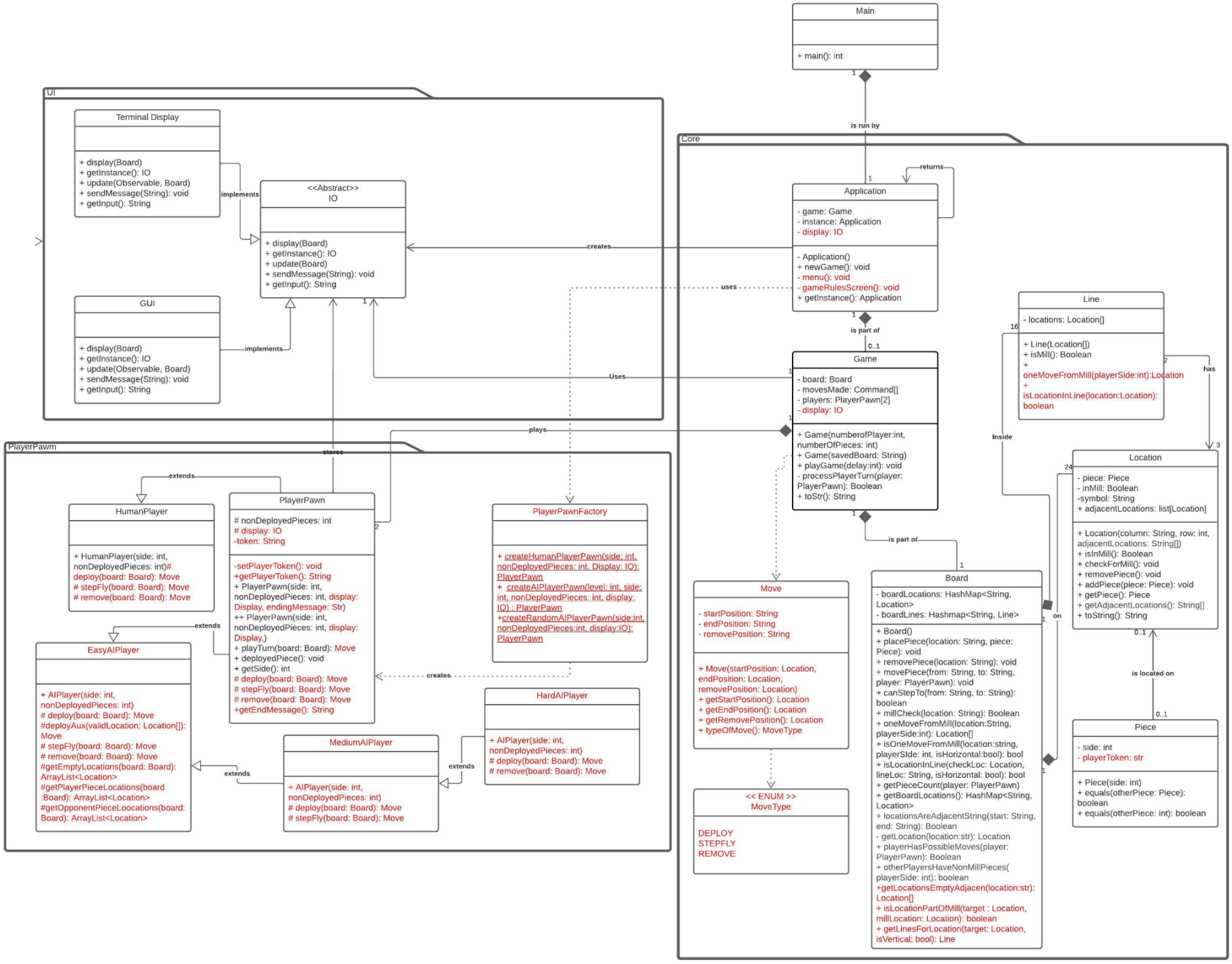
2.1. Revised Architecture

Before

(Red boxes are classes that had yet to be implemented in the previous sprint)



After
(Addition and changes are highlighted in red. Methods were added in order to easily implement AI heuristics and the general structure of the system was modified for easier implementation. For further explanation of revision choices please refer to the Architecture Revision Section)



2.2. Design Rationales

2.2.1. Advanced Requirement Design

In Sprint 1, our initial plan was to implement undo functionality, saving, and restoring functionality. However, we realized that these features wouldn't be very user-friendly in a terminal environment since there's no dedicated menu. Additionally, users would have to manually type out the file path for saving and restoring. As a result, we decided to change our advanced features and prioritize a single-player vs AI game mode instead. We believe this will be more enjoyable for the players, and it can easily be incorporated into our design.

As our initial architecture already had a decoupled and modular structure for player turns and move execution (For instance, we separated input validation and game rules from the command class, enabling easier redesign). In this design, the user input is first validated by the controller, then by the board, and finally by the game using the chain of responsibility design pattern. There were not many changes to the core functionality of the game. The main adjustments we made to our implementation were how moves are inputted into the game.

We decided to remove the command classes and instead incorporated the moves directly into the player pawn, which serves as the game controller. This modification allows moves to be executed directly by the player pawn, regardless of whether it's a human player or an AI player.

The player pawn class itself is an abstract class that was inherited from and is used as a template for the system to interact with. This allowed for player modularity as each player pawn child class handled the functionality of its moves—such as deploy, stepfly, and remove—allowing for each player to have its own heuristics as an AI or have their own method of interacting with the I/O. Allowing for reusability and expandability.

Initially, our rationale for creating command classes was to use the command behavior pattern which allows for a single point for inputting different moves into the game. Additionally, it was intended to facilitate the implementation of undo and redo functionality by saving the commands in a commands stack. However, upon further consideration, we realized that the command class would introduce unnecessary complexity, particularly when integrating AI players, due to the coupled nature of our initial command class design. By directly integrating the moves into the player pawn, we achieve a simpler and more streamlined approach. The player pawn now handles the execution of moves without the intermediary layer of command classes. This modification allows for a more flexible and straightforward implementation, especially when incorporating AI players into the game.

However, if we had applied the MVC (Model-View-Controller) architecture, we could have decoupled the command classes, allowing for the reusability of the command classes. In this case, the command classes would be responsible for different moves, while the player pawn would be responsible for selecting which commands to execute. This would provide a more flexible and modular design for our game.

2.2.2. Architecture Revision

As the advanced requirement above was being designed, the team quickly realised that changes needed to be made to the existing architecture to allow for the advanced requirement to be completed. These changes provided easier extensibility and reduced tight coupling between classes that should be more abstracted from one another.

HumanPlayer refactoring:
A large issue with our previous design were the Command classes, which are responsible for querying the I/O for input, sending some of the outputs and querying the Board to make a move. There was a different Command class for each type of move. The issue with the Command classes is that they only apply to human players, because they send output messages to the I/O and receive input messages from the I/O. This is incompatible with an AI player, which does not need I/O.

To resolve this issue, the code from the Command classes was moved into new methods in the HumanPlayer class (there is a new method for each type of move). It made sense for the code to be inside the HumanPlayer class seeing as it only applies for human players. Thus, all the Command classes were removed from our design.

AIPlayerPawn classes:
We have decided to implement different levels of AI using composition, which helps reduce code duplication and also adheres to the Liskov Substitution Principle, allowing for easy replacement of subclasses without breaking the game.

The AI levels are structured as follows:

- Deploying and moving heuristic:
    - Basic AI: This level uses random moves as its heuristic.
    - Medium AI: It tries to form mills whenever possible and then blocks opponent mills. If no strategic moves are available, it resorts to randomly selecting the next moves.
    - Hard AI: Extending the medium AI, attempting its heuristics before attempting to position the pieces strategically in locations with the most adjacent locations. In order to move pieces together for added advantage. If all else fails, it resorts to random moves.
- Removing piece heuristic:
    - Easy and medium AI: These levels randomly select opponent pieces for removal.
    - Hard AI: It checks all opponent pieces and removes the piece that would allow a mill to occur both horizontally and vertically. If such a piece doesn't exist, it removes a piece that is one move away from forming a mill. Finally, it resorts to the medium and easy implementations by randomly selecting a move.

By utilizing composition, we can build more complex heuristics on top of existing ones and utilize the lower-level heuristics as last resorts, thus reducing code duplication and increasing code reusability.

Move class:
Another issue was that the Command classes were responsible for executing moves. This gave too much responsibility to the Command classes. It makes more sense for the Game class to execute moves, since the Game class is the overall controller of the game.

To fulfil this, a Move class was created, which contains information about the move a player wishes to make. The playTurn method of the PlayerPawn class (and thus the HumanPlayer class which inherits PlayerPawn) now returns a Move class instead of executing the move inside the method. The Game class then takes the Move class and executes it. Overall, this change is a separation of concerns, allowing the player to choose a move but giving the responsibility of executing the move to the Game class.

PlayerPawnFactory class:
To address the complexity of managing multiple types of players, we introduced a PlayerPawnFactory class. This factory class is responsible for creating instances of PlayerPawn classes. With the existence of four child classes of PlayerPawn (HumanPlayer, EasyAIPlayer, MediumAIPlayer, and HardAIPlayer), the factory class simplifies the creation process for these players. It eliminates the need for higher-level classes or other classes to be aware of the specific requirements of each child class.

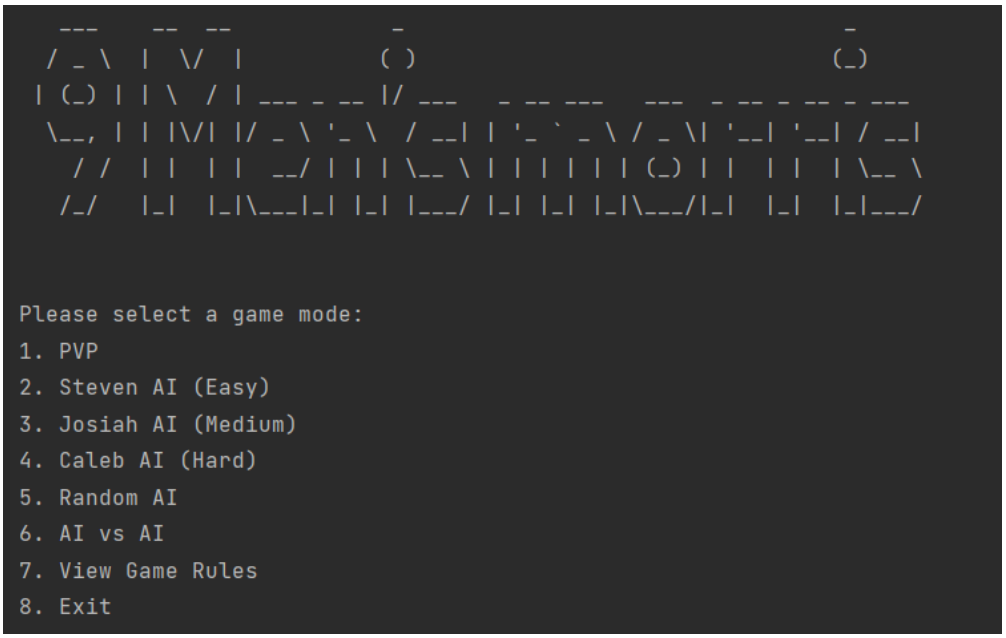
By using the PlayerPawnFactory class, the higher-level application class that initializes the players and injects them into the game no longer needs to be concerned with which child class to create or which dependencies are necessary. This decoupling allows for a more flexible design and enables the selection of random AIs. The PlayerPawnFactory class is utilized at the start of the game when the player selects a game mode to play.

The introduction of the PlayerPawnFactory class helps to manage the complexity associated with player creation, abstracting the details away from the higher-level classes and promoting a more modular and maintainable architecture.

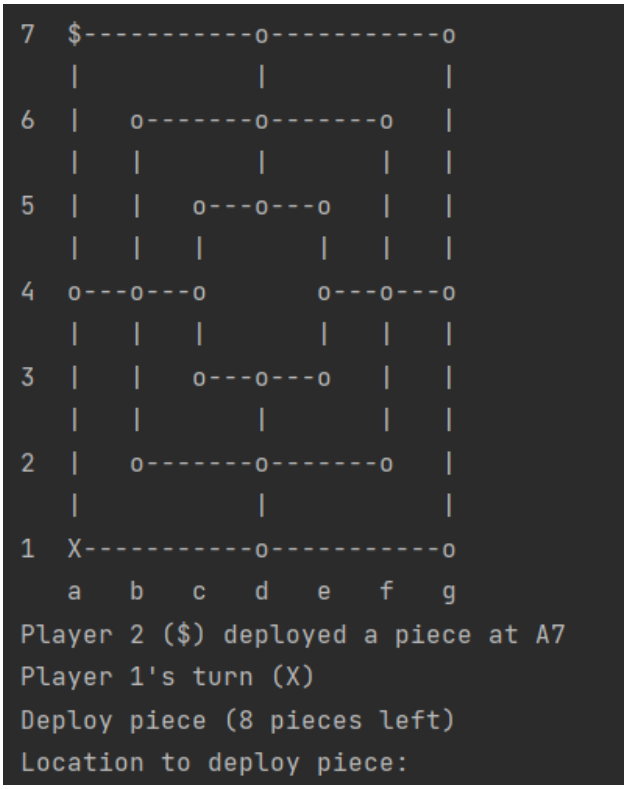
Display class:
In order to restrict the usage of the display and limit it to the controllers (human player class and the game), we made the decision to remove the singleton design from the display. Initially, we had implemented the display as a singleton, which allows for global access and potential usage throughout the application. This will potentially allow unwanted changes to be made to the display. By removing the singleton design, we ensure that the display can only be accessed by classes that require it. This decision helps to maintain a clearer separation of concerns and promotes a more controlled and structured architecture for our application.

2.2.3. Screenshots

Menu



AI (Player 2) Deploying



AI (Player 2) Forming mill and removing pieces

```
7  o-----------o-----------o
   |           |           |
6  |   o-------o-------$   |
   |   |       |       |   |
5  |   |   o---o---o   |   |
   |   |   |       |   |   |
4  o---o---o       o---$---o
   |   |   |       |   |   |
3  |   |   X---o---o   |   |
   |   |       |       |   |
2  |   X-------$-------$   |
   |           |           |
1  X-----------o-----------o
   a   b   c   d   e   f   g
Player 2 ($) deployed a piece at F6
A mill has been formed! Player 2 removed a piece at C4!
Player 1's turn (X)
Deploy piece (5 pieces left)
Location to deploy piece:
```

AI (Player 2) Moving piece

```
7  $-----------o-----------o
   |           |           |
6  |   $-------$-------$   |
   |   |       |       |   |
5  |   |   X---o---o   |   |
   |   |   |       |   |   |
4  X---X---X       o---o---$
   |   |   |       |   |   |
3  |   |   X---o---o   |   |
   |   |       |       |   |
2  |   o-------X-------$   |
   |           |           |
1  X-----------o-----------o
   a   b   c   d   e   f   g
Player 2 ($) moved a piece from F4 to G4
Player 1's turn (X)
Move piece
Location of piece to move:
```

AI (Player 2) Fly move:

```
7  $-----------$-----------o
   |           |           |
6  |   o-------o-------o   |
   |   |       |       |   |
5  |   |   X---o---o   |   |
   |   |   |       |   |   |
4  X---X---X       o---$---o
   |   |   |       |   |   |
3  |   |   X---o---o   |   |
   |   |       |       |   |
2  |   X-------o-------o   |
   |           |           |
1  X-----------o-----------o
   a   b   c   d   e   f   g
Player 2 ($) moved a piece from B6 to D7
Player 1's turn (X)
Move piece
Location of piece to move:
```

Winning

```
7  o-----------o-----------$
   |           |           |
6  |   o-------o-------o   |
   |   |       |       |   |
5  |   |   X---o---o   |   |
   |   |   |       |   |   |
4  X---X---X       o---$---o
   |   |   |       |   |   |
3  |   |   X---o---o   |   |
   |   |       |       |   |
2  |   X-------o-------o   |
   |           |           |
1  X-----------o-----------o
   a   b   c   d   e   f   g
Player 1 (X) moved a piece from D5 to C5
A mill has been formed! Player 1 removed a piece at A7!
Player 2 has less than 3 pieces! They are eliminated!
Player 1 wins!
GG
```