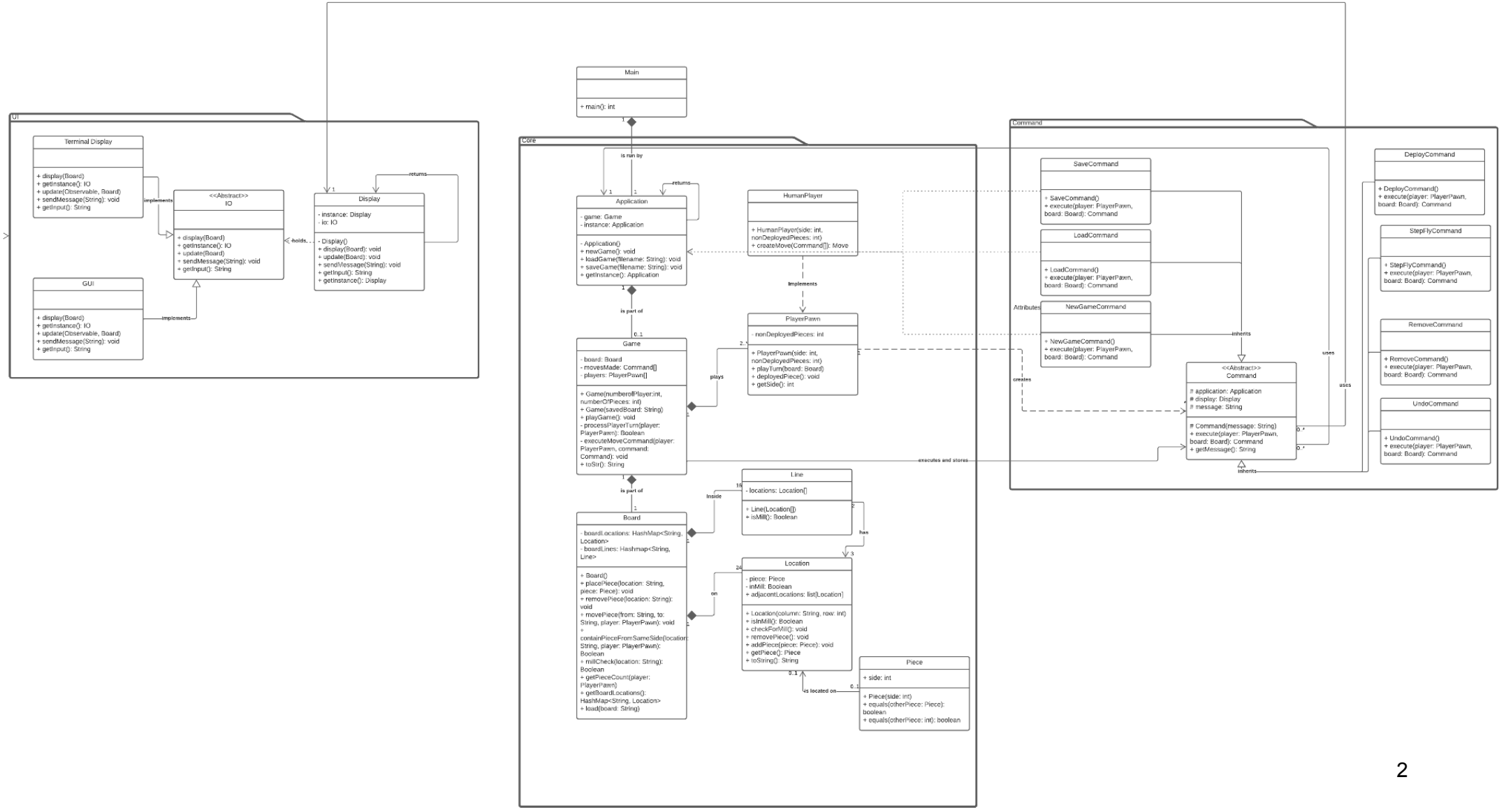
FIT3077 Sprint Three Architecture and Design Rationales Report

3 Man's Mischief (Team 38)

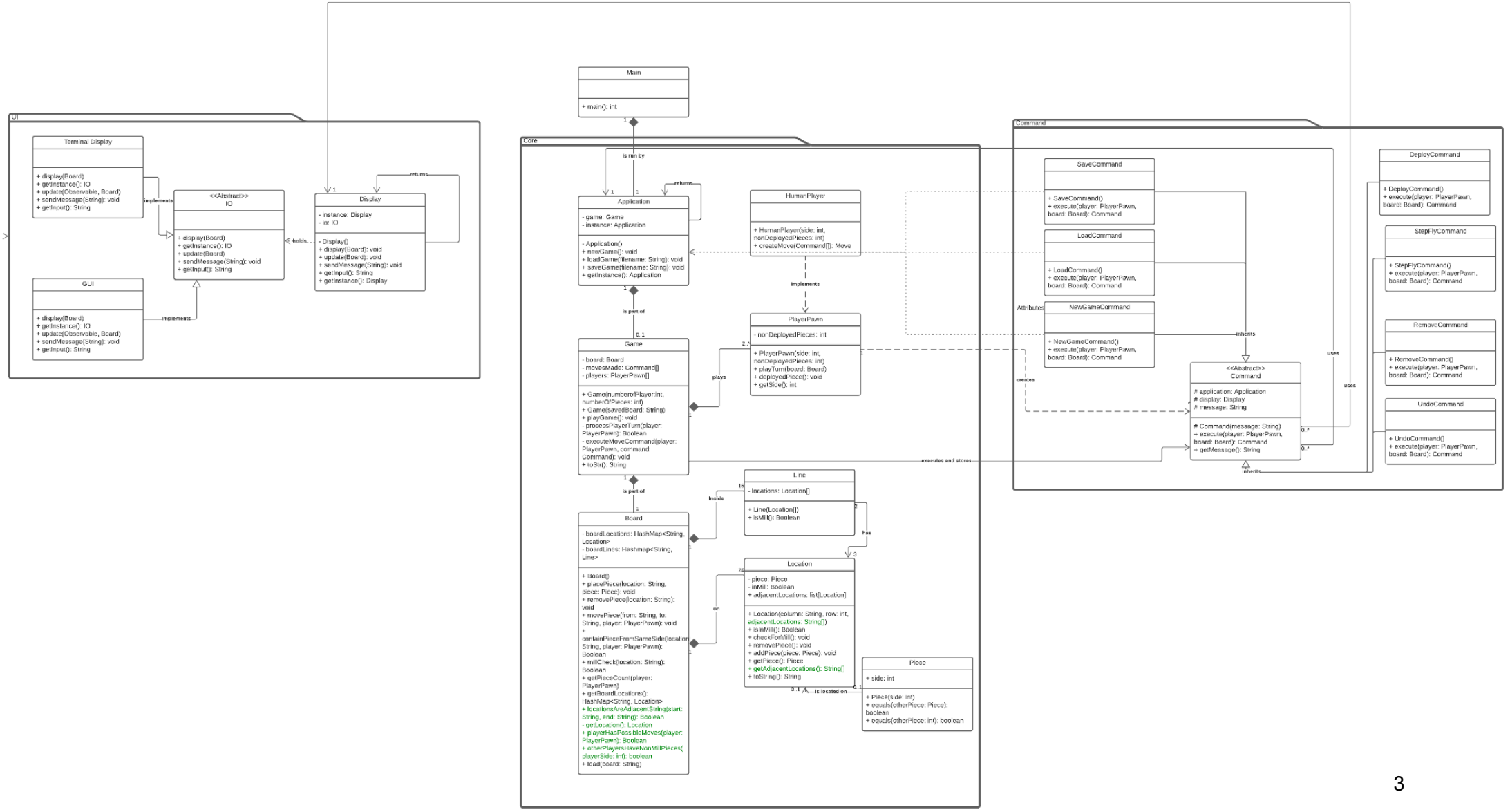
Josiah Schuller, Caleb Ooi, Steven Pham

1. Class Diagram

Before

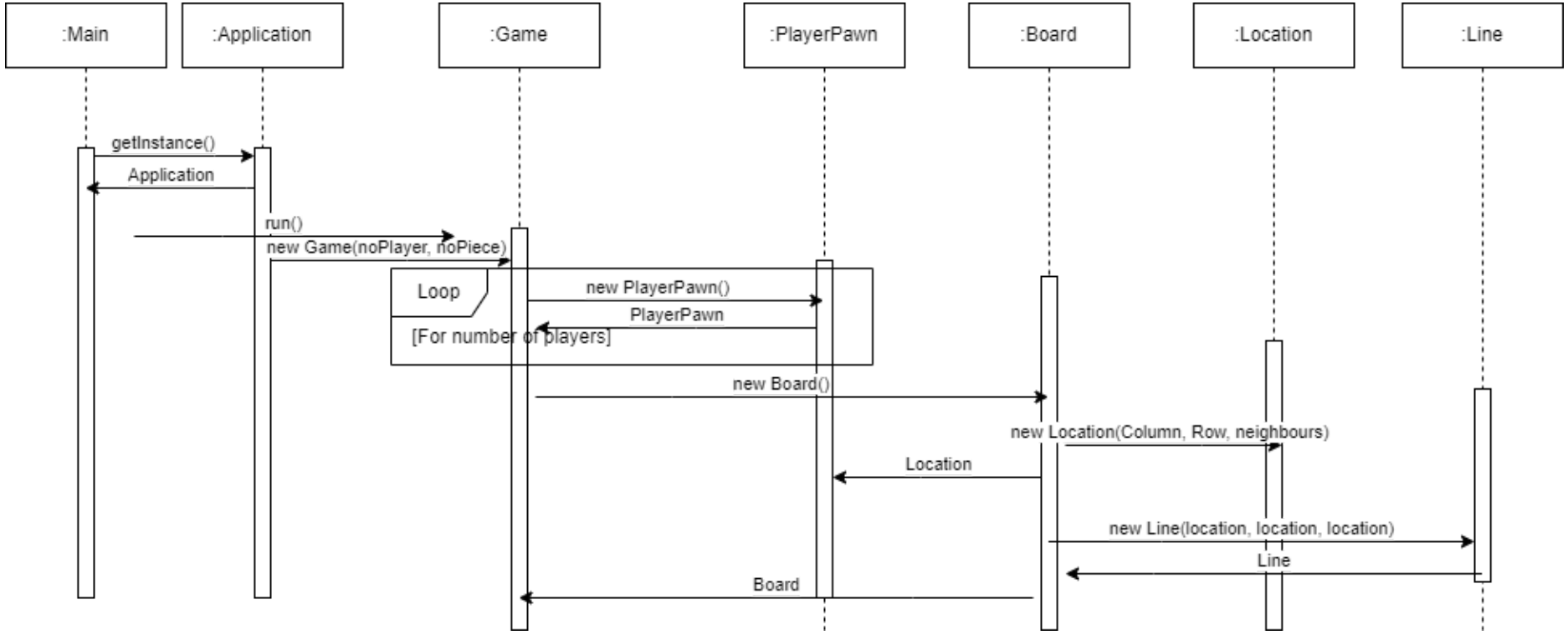


After

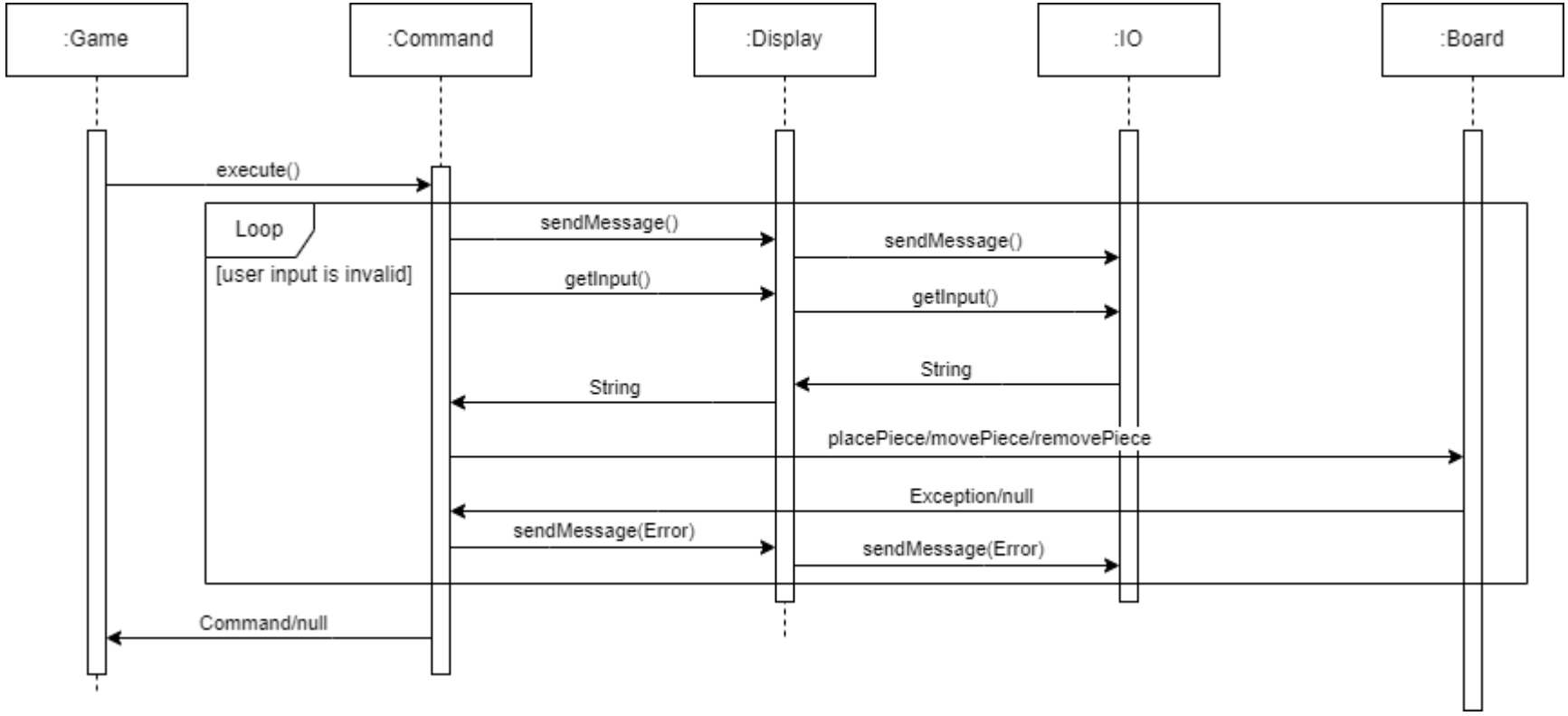

UI
Terminal Display
+ display(Board)
+ getInstance(): IO
+ update(Observable, Board)
+ sendMessage(String): void
+ getInput(): String
implements
<<Abstract>>
IO
+ display(Board)
+ getInstance(): IO
+ update(Board)
+ sendMessage(String): void
+ getInput(): String
holds
Display
- instance: Display
- io: IO
- Display()
+ display(Board): void
+ update(Board): void
+ sendMessage(String): void
+ getInput(): String
+ getInstance(): Display
returns
GUI
+ display(Board)
+ getInstance(): IO
+ update(Observable, Board)
+ sendMessage(String): void
+ getInput(): String
implements
Main
+ main(): int
is run by
Core
Application
- game: Game
- instance: Application
- Application()
+ newGame(): void
+ loadGame(filename: String): void
+ saveGame(filename: String): void
+ getInstance(): Application
returns
HumanPlayer
+ HumanPlayer(side: int, nonDeployedPieces: int)
+ createMove(Command[]): Move
implements
is part of
PlayerPawn
- nonDeployedPieces: int
+ PlayerPawn(side: int, nonDeployedPieces: int)
+ playTurn(board: Board)
+ deployedPiece(): void
+ getSide(): int
plays
Game
- board: Board
- movesMade: Command[]
- players: PlayerPawn[]
+ Game(numberofPlayer:int, numberOfPieces: int)
+ Game(savedBoard: String)
+ playGame(): void
- processPlayerTurn(player: PlayerPawn): Boolean
- executeMoveCommand(player: PlayerPawn, command: Command): void
+ toStr(): String
executes and stores
is part of
Line
- locations: Location[]
+ Line(Location[])
+ isMill(): Boolean
inside
has
Board
- boardLocations: HashMap<String, Location>
- boardLines: Hashmap<String, Line>
+ Board()
+ placePiece(location: String, piece: Piece): void
+ removePiece(location: String): void
+ movePiece(from: String, to: String, player: PlayerPawn): void
+ containPieceFromSameSide(location String, player: PlayerPawn): Boolean
+ millCheck(location: String): Boolean
+ getPieceCount(player: PlayerPawn)
+ getBoardLocations(): HashMap<String, Location>
+ locationsAreAdjacentString(start: String, end: String): Boolean
- getLocation(): Location
+ playerHasPossibleMoves(player: PlayerPawn): Boolean
+ otherPlayersHaveNonMillPieces(playerSide: int): boolean
+ load(board: String)
on
Location
- piece: Piece
- inMill: Boolean
+ adjacentLocations: list[Location]
+ Location(column: String, row: int, adjacentLocations: String[])
+ isInMill(): Boolean
+ checkForMill(): void
+ removePiece(): void
+ addPiece(piece: Piece): void
+ getPiece(): Piece
+ getAdjacentLocations(): String[]
+ toString(): String
is located on
Piece
+ side: int
+ Piece(side: int)
+ equals(otherPiece: Piece): boolean
+ equals(otherPiece: int): boolean
Command
SaveCommand
+ SaveCommand()
+ execute(player: PlayerPawn, board: Board): Command
LoadCommand
+ LoadCommand()
+ execute(player: PlayerPawn, board: Board): Command
Attributes
NewGameCommand
+ NewGameCommand()
+ execute(player: PlayerPawn, board: Board): Command
creates
inherits
<<Abstract>>
Command
application: Application
display: Display
message: String
Command(message: String)
+ execute(player: PlayerPawn, board: Board): Command
+ getMessage(): String
uses
inherits
DeployCommand
+ DeployCommand()
+ execute(player: PlayerPawn, board: Board): Command
StepFlyCommand
+ StepFlyCommand()
+ execute(player: PlayerPawn, board: Board): Command
RemoveCommand
+ RemoveCommand()
+ execute(player: PlayerPawn, board: Board): Command
UndoCommand
+ UndoCommand()
+ execute(player: PlayerPawn, board: Board): Command

2. Sequence/Communication diagrams

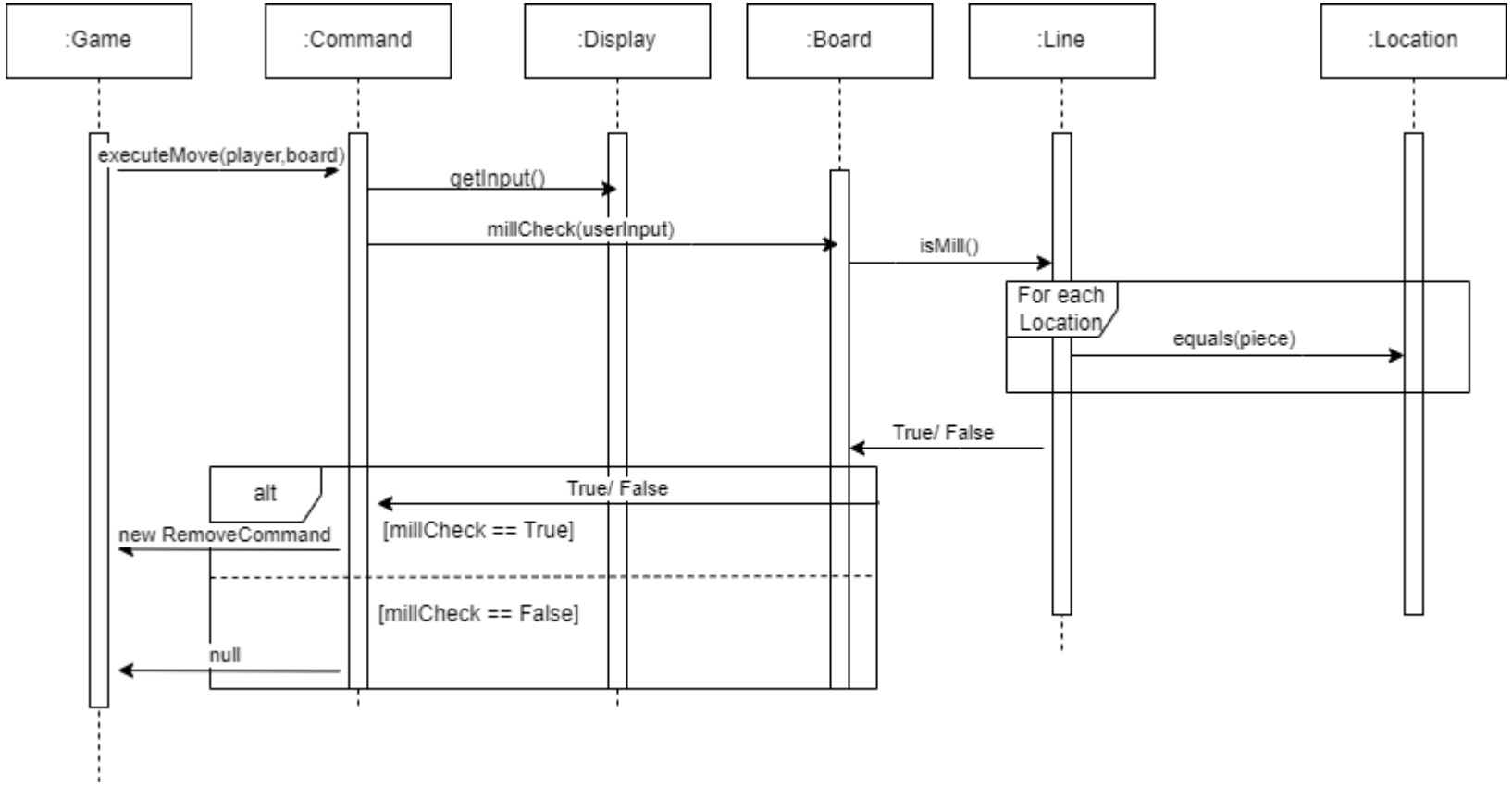
2.1 Game initialisation



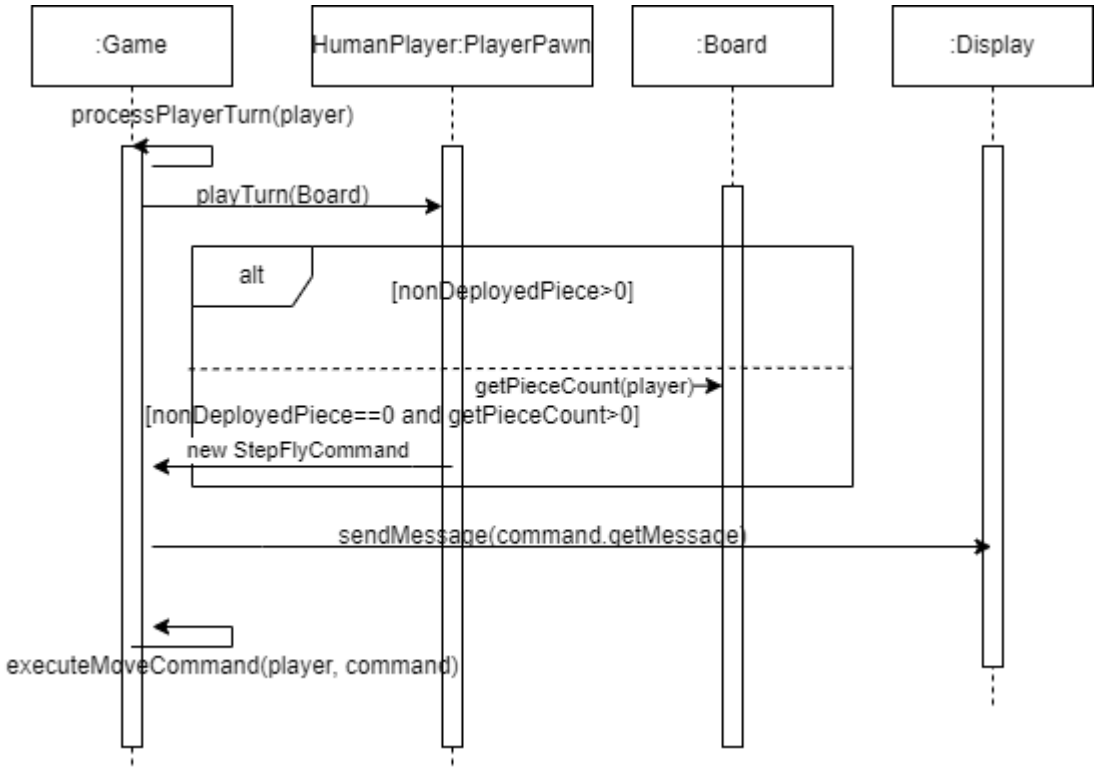
2.2 Getting and validating user input



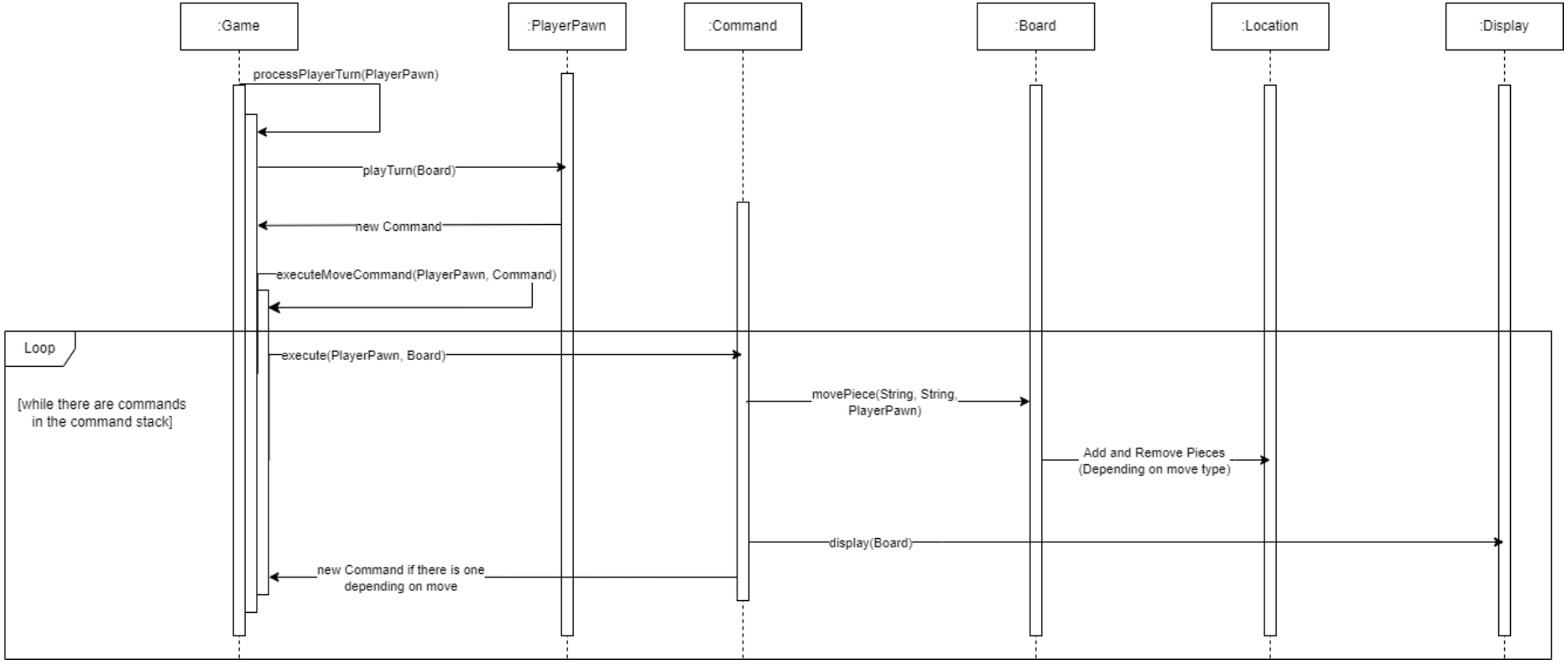
2.3 Determining if there is a mill



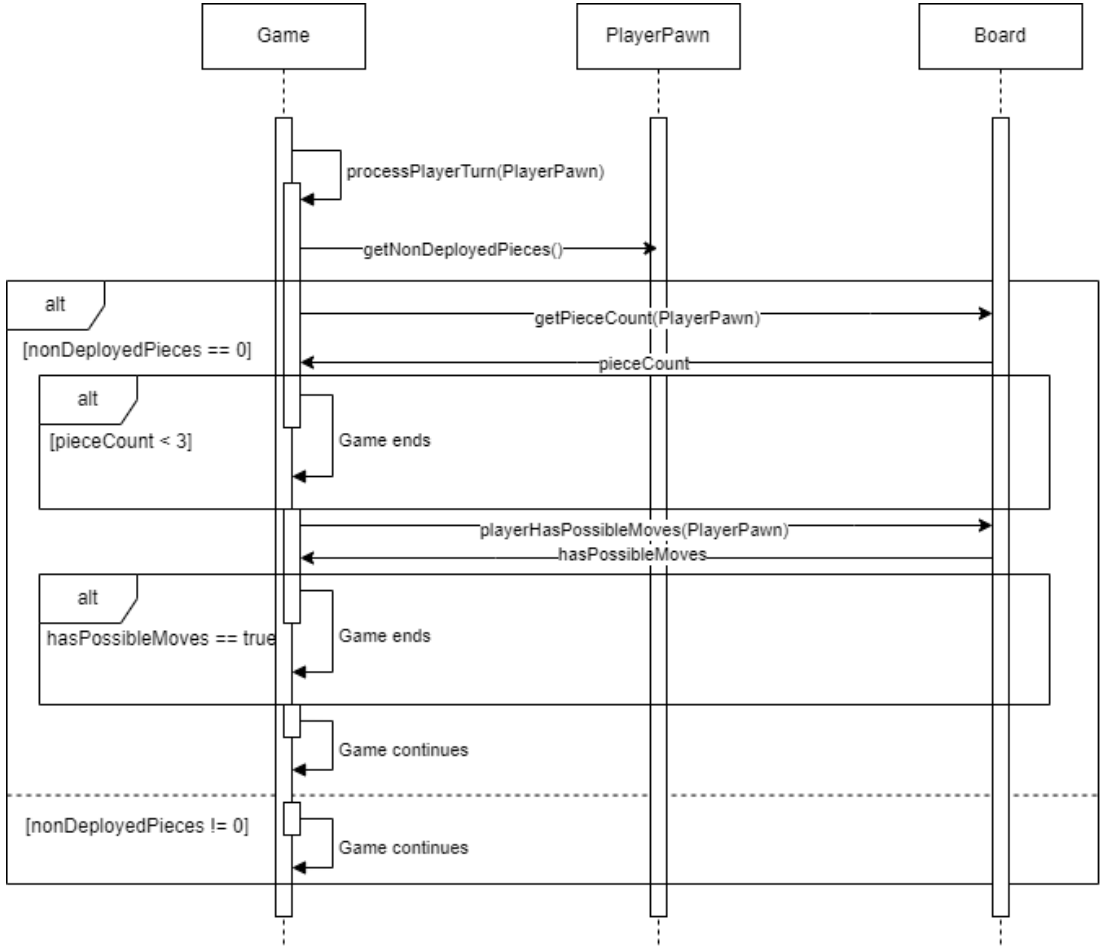
2.4 Determining whether a move is valid



2.5 Executing a move



2.6 Game ending



3. Design Rationales

3.1 Revised architecture

For the class diagram, additions have been highlighted in green and removals have been highlighted in red.

The core structure of the architecture has stayed the same as the original design has allowed for the implementation of the necessary features of the game and changes such as the removal or addition of classes are not necessary.

A couple of methods have been added to a couple of classes in order to implement features of the game such as *only being able to step to adjacent locations*, *not being able to remove milled pieces*, and *winning the game*. A method to check if two locations are adjacent were added to Board and Location (*getAdjacentLocations* and *locationsAreAdjacentString*) and a way to scan if a player has lost if they have no moves they can make (*playerHasPossibleMoves*). An additional check was added to removing pieces where it first checks if the piece that is to be removed is in a mill (*otherPlayersHaveNonMillPieces*).

Overall, the design has remained the same as change was not necessary as the full implementation of the game had already been accounted for in the previous sprint's design.

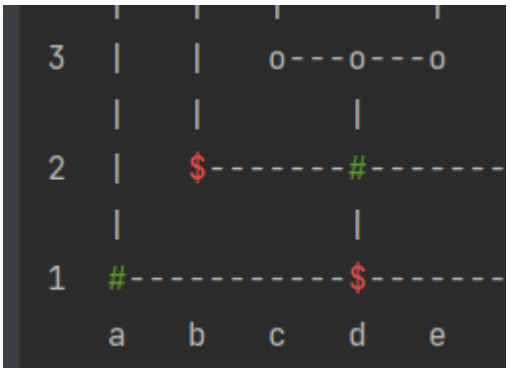
3.2 Quality attributes

3.2.1 Usability

As in any piece of software, usability is essential to ensure that any user is able to complete what they want to do in a timely manner and be satisfied by the end of the experience. Usability is important in 9MM so that the users have an enjoyable experience while playing the game. Here are two examples of ways usability has impacted our design:

- Colours and colour-blind accessibility:
  Colours are used to signify which side each token is on. Colours make it easier for users to discern the board at a single glance. But colours are not the only way to discern which side a token is on. Each side is also represented by a different symbol, so that colour-blind users can still identify tokens from different sides. See the

screenshot below:



- Error prevention:
  Users make errors - all the time. To make the user experience as pleasant as possible, we as a development team need to ensure that the application prevents the users' errors and warns them if they do make an error (instead of the game simply crashing). In 9MM, we apply sanitation checks to any user input to ensure that it is a valid input. For example in the screenshot below, the user attempts to move a piece from an invalid square and then attempts to move an opponent's piece. Each time, the game informs the user on the mistake they have made and gives the user another chance to give a correct input.

```
Player 1: Move piece
Location of piece to move:
askdjflkadjs
Location does not exist
Location of piece to move:
b4
Invalid location! You can only move your own piece
Location of piece to move:
```

3.2.2 Portability

Portability is crucial in making software accessible and convenient for users to play the game. The game was designed with this in mind through careful choice of a stack that allowed the game to be as portable as possible. By programming the game in Java, it allows it to be easily ported to other platforms through the use of its interpreter, Java Virtual Machine (JVM), which allows the game to run on multiple platforms independent of the implementation. Making our game portable.

3.2.3 Performance Efficiency

Performance is a crucial aspect when designing the game, ensuring that players can enjoy smooth gameplay and responsive user interactions. We put a focus on creating an efficient method to handle user input validation and retrieve the input location.

To achieve this, an optimised hashmap data structure is employed, specifically designed for constant-time reading. This enables quick validation of user input by utilising the input as a key. If the key is valid, the location associated with it is retrieved seamlessly. In case of an invalid key, an exception is thrown and caught by the command, ensuring minimal delay between the user's action and the game's response.

3.3 Human values

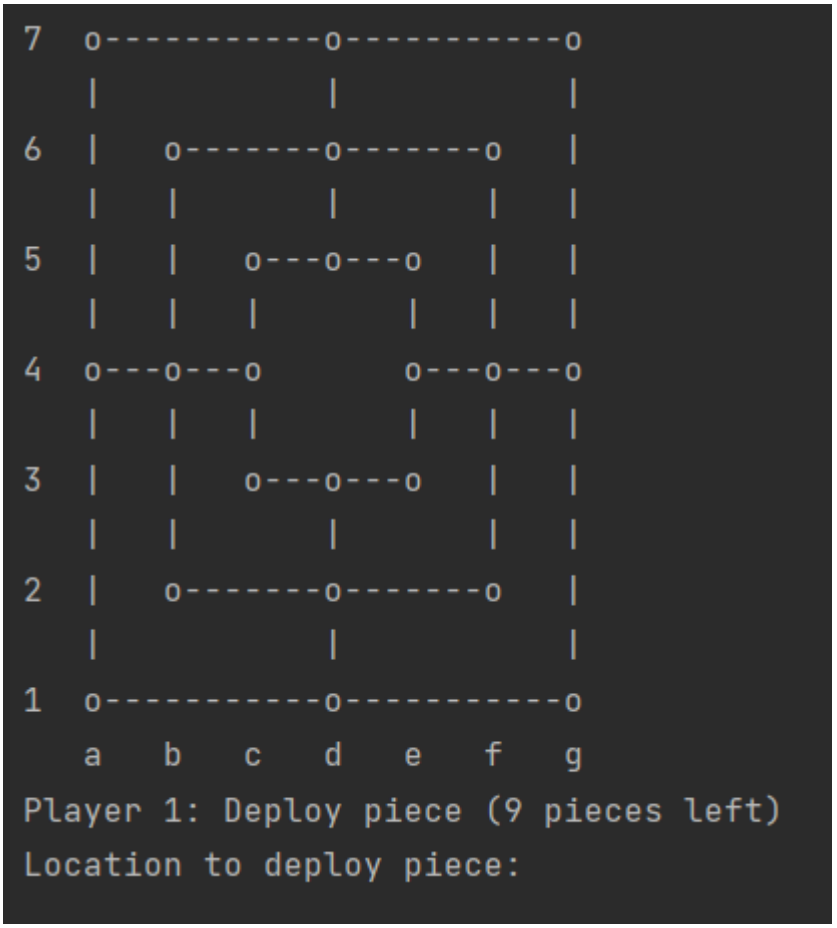
A key human value we have incorporated into our design is the value of self-direction and control. In our design, we give the user the necessary information they need to play the game (i.e. the board, prompts for input, how many tokens the user has left to place, etc.), but we do not pamper them with any more information than that. The user interface is clean and gives the user the maximum freedom possible so that they can do what they want (without breaking the rules of the game of course).

Another key human value we have incorporated into our design is the value of Universalism. Our game utilised a platform independent programming language, java. The versatility of java and its Java Virtual Machine enable the game to be accessible to a wide range of users across different operating systems and devices. Another aspect of universalism is that the game design has a highly fault-tolerant input which accommodates lower and upper case input. These elements work together to provide an accessible and inclusive gaming experience for users across different platforms and input preferences.

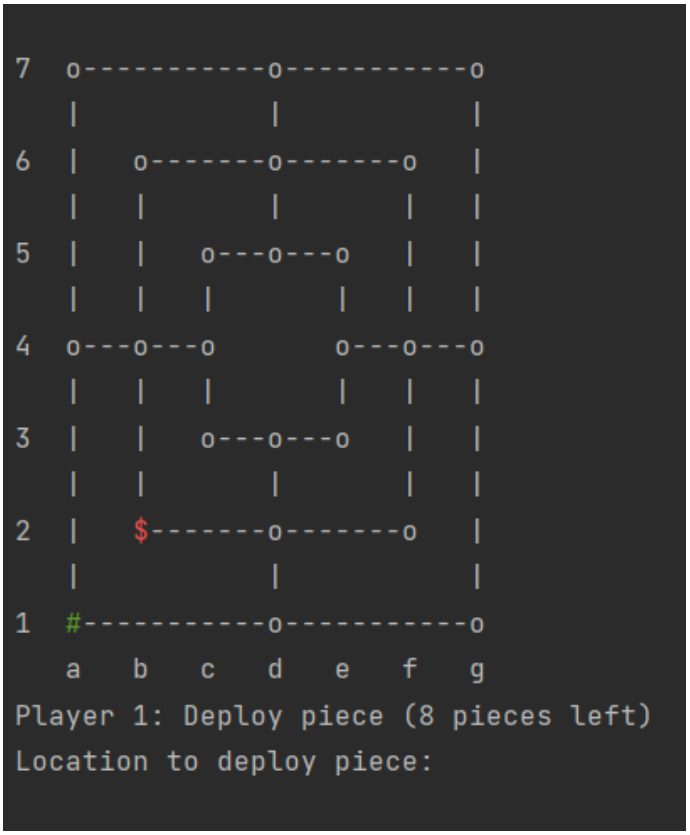
Another key human value accounted for in our design would be pleasure. The core purpose of any game is for the player to have fun. And by making the game as quickly accessible, it allows the player to be able to engage with the game as soon as possible and have fun. This was achieved in our design by making the UI simple but clear and how you interact with the game, the input, simple. There is a lack of ambiguity with the UI as only what is necessary is displayed, the board and the input lines at the bottom. With input, the user is prompted with what to do so there is no ambiguity. This makes the game more accessible due to a very low learning curve and thus helps achieve the human value of pleasure.

4. Screenshots of different Game States

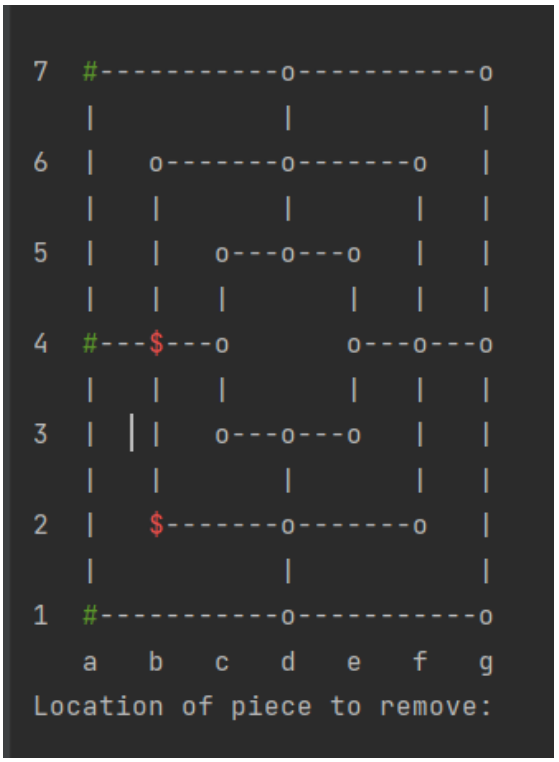
4.1 Game initialisation- Player 1 starts with 9 pieces

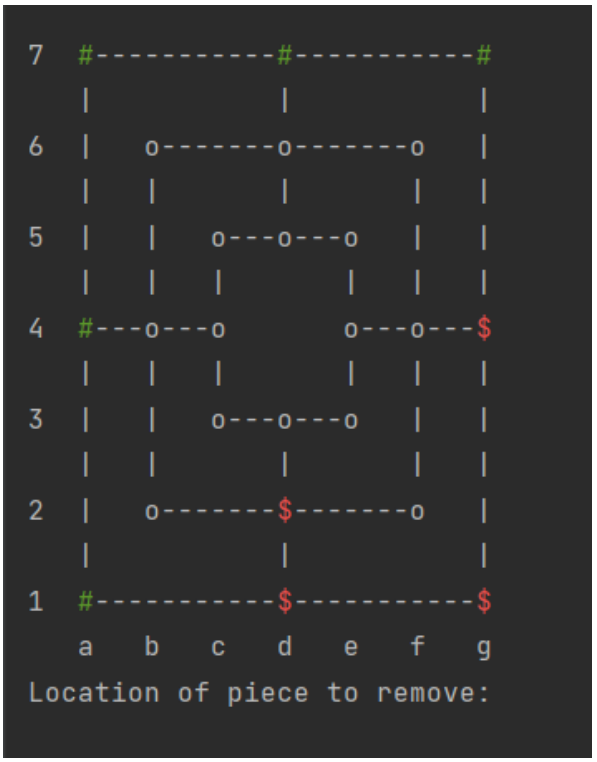


4.2 Deploying



Mill detection- Players are able to remove 1 piece when one or more mills occur.





4.3 Removing State -A piece can only be removed if its not in a mill

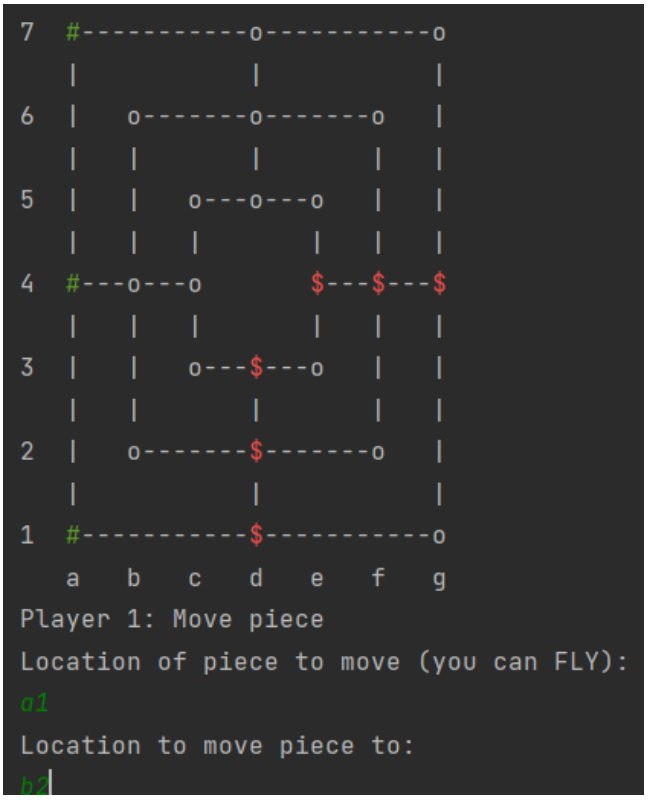
```
7  #-----------#-----------#
   |           |           |
6  |   o-------o-------$   |
   |   |       |       |   |
5  |   |   o---o---o   |   |
   |   |   |       |   |   |
4  #---#---#       $---$---$
   |   |   |       |   |   |
3  |   |   o---$---o   |   |
   |   |       |       |   |
2  |   o-------$-------o   |
   |           |           |
1  #-----------$-----------$
   a   b   c   d   e   f   g
Location of piece to remove:
d3
Invalid location! You cannot only remove a piece in a mill
Location of piece to remove:
```

Moving State- Players are able to move their pieces on the board to adjacent location when all the pieces have been deployed

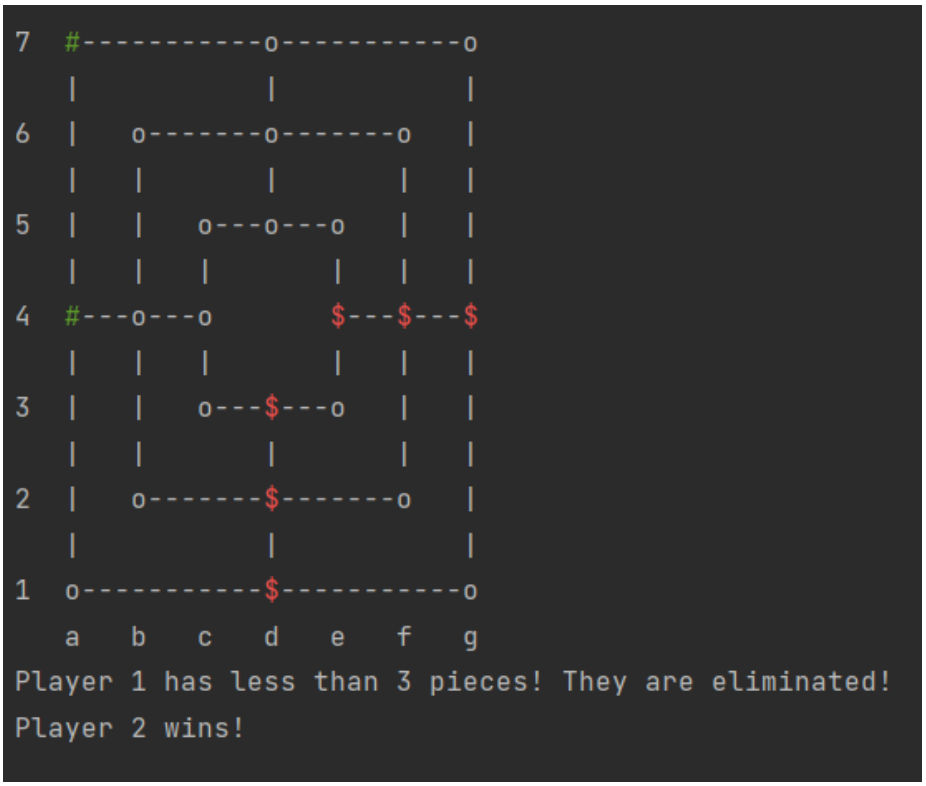
```
7  #-----------#-----------#
   |           |           |
6  |   o-------#-------$   |
   |   |       |       |   |
5  |   |   o---o---#   |   |
   |   |   |       |   |   |
4  #---#---o       $---#---$
   |   |   |       |   |   |
3  |   |   o---o---$   |   |
   |   |       |       |   |
2  |   o-------$-------$   |
   |           |           |
1  #-----------$-----------$
   a   b   c   d   e   f   g
Player 1: Move piece
Location of piece to move:
a6
Location does not exist
Location of piece to move:
d6
Location to move piece to:
d5
```

```
7  #-----------#-----------#
   |           |           |
6  |   o-------o-------$   |
   |   |       |       |   |
5  |   |   o---#---#   |   |
   |   |   |       |   |   |
4  #---#---o       $---#---$
   |   |   |       |   |   |
3  |   |   o---o---$   |   |
   |   |       |       |   |
2  |   o-------$-------$   |
   |           |           |
1  #-----------$-----------$
   a   b   c   d   e   f   g
Player 2: Move piece
Location of piece to move:
e4
Location to move piece to:
d3
You can only fly when you have 3 pieces or less
Location of piece to move:
```

4.4 Flying State - Players can only fly when theres 3 or less pieces on the board



4.5 Winning- Player wins if opponent has less than 3 pieces



4.6 Winning- Player wins if opponent has less than 3 pieces