FIT3077 Sprint One Report

Team: MA_Friday2pm_Team10
Team Name: The Algorithm Alchemists

Team Members:
Lai Qui Juin 32638809
Leong Yen Ni 32023685
Lim Jia Ying 32423063
Lo Kin Herng 32023995

Table of Contents

1. Team Information

1.1 Team Name and Team Photo

- Team Name: The Algorithm Alchemists
- Team Photo:

From left to right: Jia Ying, Yen Ni, Qui Juin, Kin Herng

1.2 Team Membership

Name	**Student ID**	**Email**	**Contact No.**
Lai Qui Juin	32638809	qlai0003@student.monash.edu	012-4914509
Leong Yen Ni	32023685	yleo0013@student.monash.edu	012-3523168
Lim Jia Ying	32423063	jlim0165@student.monash.edu	012-6882261
Lo Kin Herng	32023995	kloo0013@student.monash.edu	018-2132787

Name	**Technical & Professional Strengths**	**Fun Fact**
Lai Qui Juin	Front-end	I enjoy washing dishes.
Leong Yen Ni	Front-end	I like to drink milk.
Lim Jia Ying	Java	I reduce stress by eating good food.
Lo Kin Herng	OOP Design	I cannot ride a bike.

1.3 Team Schedule

- Regular meeting: Friday 4-6pm (Physical)
- Regular work schedule: Expected 8 hours of work per week per member
- Workload distribution:
    - Task allocation will be done in a Trello board that is shared among the team members. (Trello link : https://trello.com/invite/b/4CLM3s40/ATTI48baff80602aca79b3d0d86951d906e42602BF77/fit3077-the-algorithm-alchemists)
    - User stories will be added as cards in the Trello board.
    - Each team member will be responsible for some user stories.
    - Google Drive shared folder is used to store all documents.
    - Reports and documentation will be done together by all members using Google Docs.

1.4 Technology Stack and Justification

Programming language and framework	JavaScript Front-end: React.js Back-end: TypeScript	Java Front-end: JavaFx & Scene Builder Back-end: Java
Availability of resources	React has a larger community and wider range of libraries	JavaFx has a relatively small community and lesser libraries
UI library	React is a library for building user interfaces, requires additional tools and libraries for other features	JavaFx provides a complete UI framework with a rich set of UI controls and layouts
Type of application	Web Application	Java Native App
App performance	Web apps compiles and runs slower than native applications	Native apps compiles and runs faster, more efficient
Industrial usage	React is widely used in the industry	JavaFx is less popular in the industry
Developing performance	Virtual DOM allows React to efficiently update the UI without re-rendering the entire page	Re-rendering required if modifications to UI are made when coding in JavaFx
Reusability	React components can be easily reused across the application	JavaFx supports the creation of custom UI components which can be reused across the application
Ease of use	React can only be implemented by coding	JavaFx can be implemented by drag and drop using Scene Builder
Integration	API is required to connect React front-end with TypeScript back-end	JavaFx integrates well with other Java libraries and frameworks
Cost of maintenance	React and TypeScript has a simpler and more concise syntax compared to Java	Java has a longer syntax

Team experience	Less experience in coding OOP with TypeScript	All members are experienced in coding OOP with Java

Based on the table above, our team has considered developing our game using React.js + Typescript or JavaFx + Java. JavaFx provides a complete UI framework with a rich set of UI controls and layouts. If we are using JavaFx, we will need to develop our game as a native app which compiles and runs faster. Additionally, JavaFx supports the creation of custom UI components which can be reused across the application and it can be implemented by drag and drop using Scene Builder. Furthermore, JavaFx integrates well with other Java libraries and frameworks. Lastly, all of our team members have taken OOP units previously and are comfortable with coding OOP using Java compared to TypeScript.

In conclusion, JavaFx and Java are our chosen tech stack for front end and back end development respectively. Note that we are still open to changes and may include more technologies into our tech stack further down the road.

2. User Stories

2.1 Advanced requirements chosen

b. Players are allowed to undo their last move and the game client should support the undoing of moves until there are no more previous moves available. The game client also needs to be able to support saving the state of the currently active game, and be able to fully reload any previously saved game(s). The game state must be stored as a simple text file where each line in the text file represents the current state of the board and stores information about the previously made move. It is anticipated that different file formats will be required in the future so any design decisions should explicitly factor this in.
c. 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.

2.2 Definitions

1. Token - Each player has 9 tokens (men)
2. Mill - A straight row of 3 tokens along the board's lines (i.e. not diagonally)
3. Positions - 24 line intersections on the game board

2.3 User stories

Game Player

1. As a game player, I want to be able to choose to play with **another player or computer** so that the game is playable as the game requires 2 players.
2. As a game player, I want to **place** my token on an **empty** position on the game board so that I can form a mill and avoid the opponent from forming a mill.
3. As a game player, I want to **move** my token to an **empty adjacent** position on the board so that I can form a mill and avoid the opponent from forming a mill.
4. As a game player, I want to **move** my token to any **empty** positions on the board when I am still left with three tokens so that I can form a mill and avoid the opponent from forming a mill.
5. As a game player, I want to **remove** my opponent's token that is not part of a mill when I form one, unless no other tokens can be removed, so that I can be one step closer to winning.
6. As a game player, I want to be able to **undo** my moves so that I can perform a different move if I accidentally made a wrong move.
7. As a game player, I want to be able to **save** any unfinished games so that I can continue playing those games in the future.
8. As a game player, I want to fully **reload** any previously saved game so that I can resume playing the game.
9. As a game player, I want to be able to **quit** the game so that I can stop playing and return to the main page.

Game Board

1. As a game board, I want to make sure both players **start off with 9 tokens** each so that the game is fair for both players.

2. As a game board, I want to make sure both players **place all their tokens** so that the players can start moving their tokens.
3. As a game board, I want to have tokens of **two different colours** so that I can differentiate which token belongs to each player.
4. As a game board, I want to start off with **24 empty positions** so that the players can place tokens onto the board.
5. As a game board, I want to make sure both players **take turns to play** so that the game is fair.
6. As a game board, I want to make sure that only **1 token** can be placed at **1 position** at one point in time so that tokens cannot be stacked.
7. As a game board, I want to make sure that each player can only **place/move 1 token** at a time so that the game is fair.
8. As a game board, I want to check if a player has **2 tokens or less** so that the game can end.
9. As a game board, I want to check if a player **no longer has any legal moves left** so that the game can end.
10. As a game board, I want to display an **alert** when a player tries to **remove** an opposing player's token when they **did not just form a mill** so that the game rule is not violated.
11. As a game board, I want to display an **alert** when a player tries to **move his piece before** they have **placed all their pieces on the board**.
12. As a game board, I want to display an **alert** when a player with more than 3 tokens tries to move to a **non-adjacent** position so that the game rule is not violated.
13. As a game board, I want to display an **alert** when a player tries to **remove** a token that is **from a mill** when there is a token that is not part of a mill so that the game rule is not violated.
14. As a game board, I want to display an **alert** when a player tries to place or move a token to a **non-empty adjacent** position so that the game rule is not violated.
15. As a game board, I want to be able to **announce** the **outcome** of the game when a win condition has been met so that the players know who has won and the game can be ended.
16. As a game board, I want to **save the state** of the current active game in a text file at each turn so that the information about the previously made move can be stored and players can undo their moves.

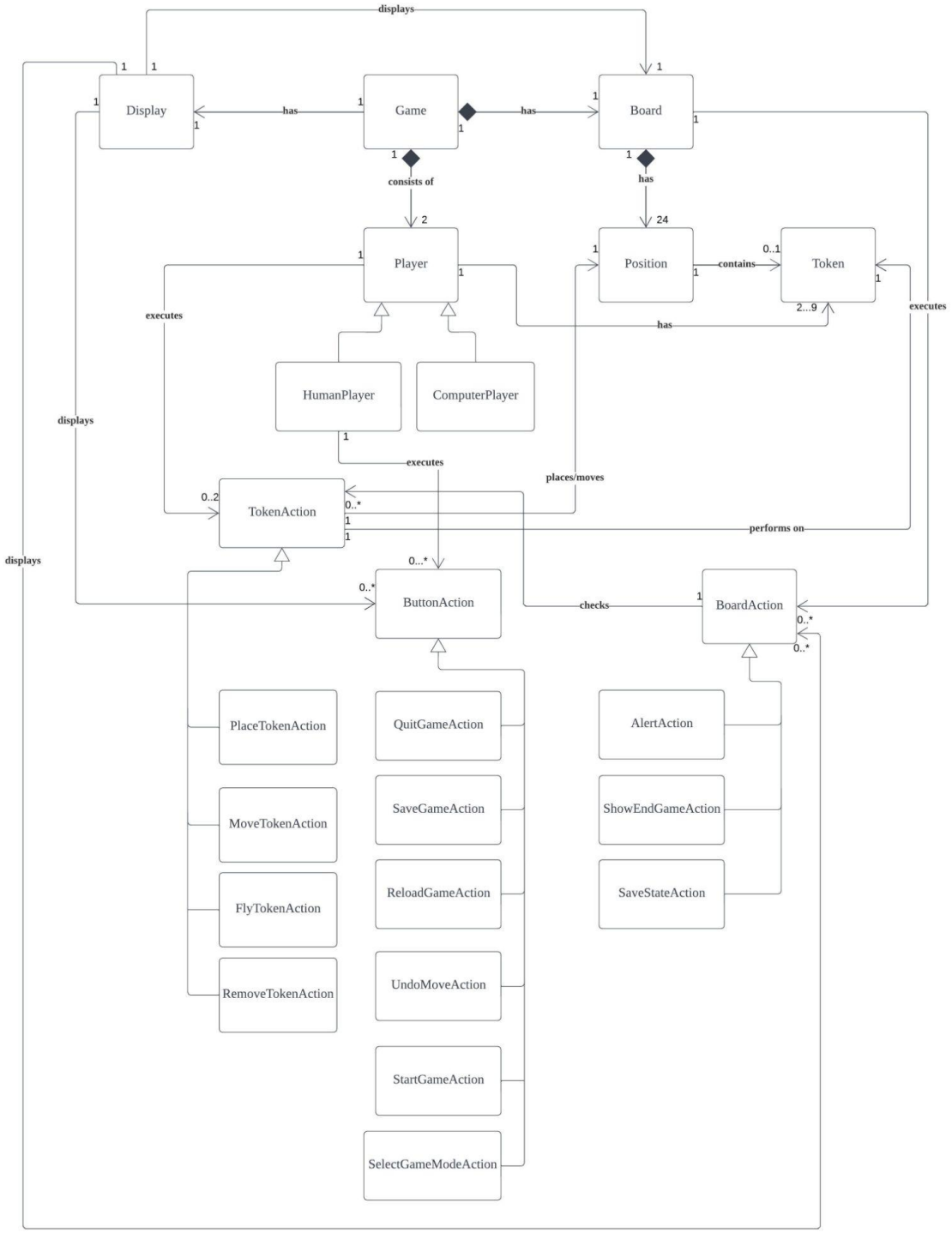
Computer (Opponent)

1. As a computer, I want to be able to place a token **randomly** in an empty position on the game board, so that I can form a mill.
2. As a computer, I want to be able to make **random** moves to move my token to an **adjacent** empty position so that I can continue to play the game with the other player.
3. As a computer, I want to be able to make **random** moves to move my token to any **empty** position when I have 3 tokens left so that I can continue to play the game with the other player.
4. As a computer, I want to **remove** one of the opponent's tokens randomly when I form a mill, so that I can be one step closer to winning.

3. Basic Architecture

3.1 Domain Model

Domain Model Link:
https://lucid.app/lucidchart/e66943cd-742a-4ce3-8dd3-ba4812ec3b4b/edit?invitationId=inv_52e98539-aab1-4b23-8d36-c80cdd648ae7&page=KABKOsk9gNH1#



3.2 Justifications

The Nine Men's Morris game is played by 2 players and requires a board with 24 positions which are the line intersections, and 9 tokens for each player. We have first identified several domain entities such as Game, Board, Positions, Token and Player. With that, we have established the relationships between these entities:

1. **Game has a composition relationship with the Board**, as the game board is a major part of the game. Without the game, the game board cannot exist on its own.
2. **Board has a composition relationship with Position**. Similarly, the positions on the board exist together with the game board, it is impossible for the positions to exist independently.
3. Each position on the board can accommodate either one or zero tokens, indicating **an association between the Position and Token.**
4. As the game requires exactly 2 players, there is a **one-to-many composition relationship between the Game and Player**. This relationship ensures that a Player cannot exist without being associated with a Game entity, and that the Game entity cannot exist without exactly two Player entities.
5. Each player is assigned a total of 9 tokens to play the game, which means there is an **association between Player and Token** entities. The multiplicity between these two is 1 to 2...9, because having less than 2 tokens means that the player has lost and the game will end.

The Player entity mentioned above will act as an abstract class to allow generalisation for other entities. This is necessary to implement the advanced requirement of allowing a single player to play the game against the computer. For that, the game now has two types of players, the human player and the computer player. To achieve this, we have created **HumanPlayer and ComputerPlayer entities that inherit from the Player entity**.

Moving on to the game mechanics, players take turns placing, moving, and removing tokens from the game board. Thus, we have created three entities to model these actions, which are PlaceTokenAction, MoveTokenAction, and RemoveTokenAction. However, directly associating these actions with the Player entity would violate the **Open-Closed Principle**. Instead, we introduce a TokenAction abstract class to allow for extension without modification. The following relationships exist between the entities:

1. **PlaceTokenAction, MoveTokenAction, FlyTokenAction and RemoveTokenAction** inherit from **TokenAction**. We create an entity for each of these identified actions to obey the **Single Responsibility Principle** because each entity has its own responsibilities:

PlaceTokenAction	Allow players to place a token on board.
MoveTokenAction	Allow players to move a token to an empty adjacent position.
FlyTokenAction	Allow players to move a token to any empty position when the player has 3 tokens left.
RemoveTokenAction	Allow players to remove the opponent's token when the player forms a mill.

2. **TokenAction is associated with Player** because we need to know which player is performing the action. A player usually makes 1 move per turn, but may make 0 moves if he decides to stop playing, and may make 2 moves when he moves a token to form a mill and then removes the opponent's token. This is the rationale for the multiplicity of this relationship.
3. **TokenAction is associated with Position** because we need to know the initial position and destination position when placing/moving the tokens.
4. **TokenAction is associated with Token** because we need to know which token the action is being performed on.

Furthermore, certain actions in the game are specific to HumanPlayers only, which can be performed by clicking on certain buttons on the user interface to accomplish different tasks, which include the advanced requirements of undoing moves, saving and reloading previously played games. The following are the entities that were identified:

StartGameAction	Allow players to start a new game
SelectGameModeAction	Allow players to select between 2-player mode or playing against the computer
SaveGameAction	Allow players to save a game and continue playing later
ReloadGameAction	Allow players to continue playing an unfinished game by reloading it
UndoMoveAction	Allow players to undo their last move until no previous moves are available
QuitGameAction	Allow players to exit the game and return to the main page

Following the same reasoning as above, a **ButtonAction** entity will act as an abstract entity allowing the six action entities above to **inherit from it**. As a human is required to click these buttons**, HumanPlayer** is also **associated with the ButtonAction entity.** A human can click 0 or more buttons, which explains the multiplicity of the relationship.

The Board is responsible for governing the rules of the games. It keeps track of the moves made by players to ensure no rules are violated and to check for the win condition so that the game can end. Thus, BoardAction is an abstract entity that represents the roles that the board must take on. **BoardAction has an association with TokenAction** because the board must keep track of moves made, and **BoardAction has an association with BoardAction** because the Board has to execute those actions. Similar to the justification for TokenAction and ButtonAction above, a BoardAction entity will be a parent entity to allow the three board action entities to inherit from it. The following are the board actions identified:

AlertAction	Detect any illegal moves made and prevent player from making the move by showing a pop up alert to the player
ShowEndGameAction	Checks for the win conditions and ends the game by announcing the winner
SaveStateAction	Saves the state of the game after every player's move to support the undo function

Lastly, we need a **Display** entity to represent the game's interface for user interaction. The following entities are **associated** with the Display:

1. The **Game** and **Board** entities are displayed on the screen for the players to view and interact with.
2. **ButtonAction** entities require players to click on a button to perform an action, so buttons must be displayed on the interface.
3. After a **Board** executes a BoardAction, a pop-up notification appears on the screen to inform players of certain messages.

4. Basic UI Design

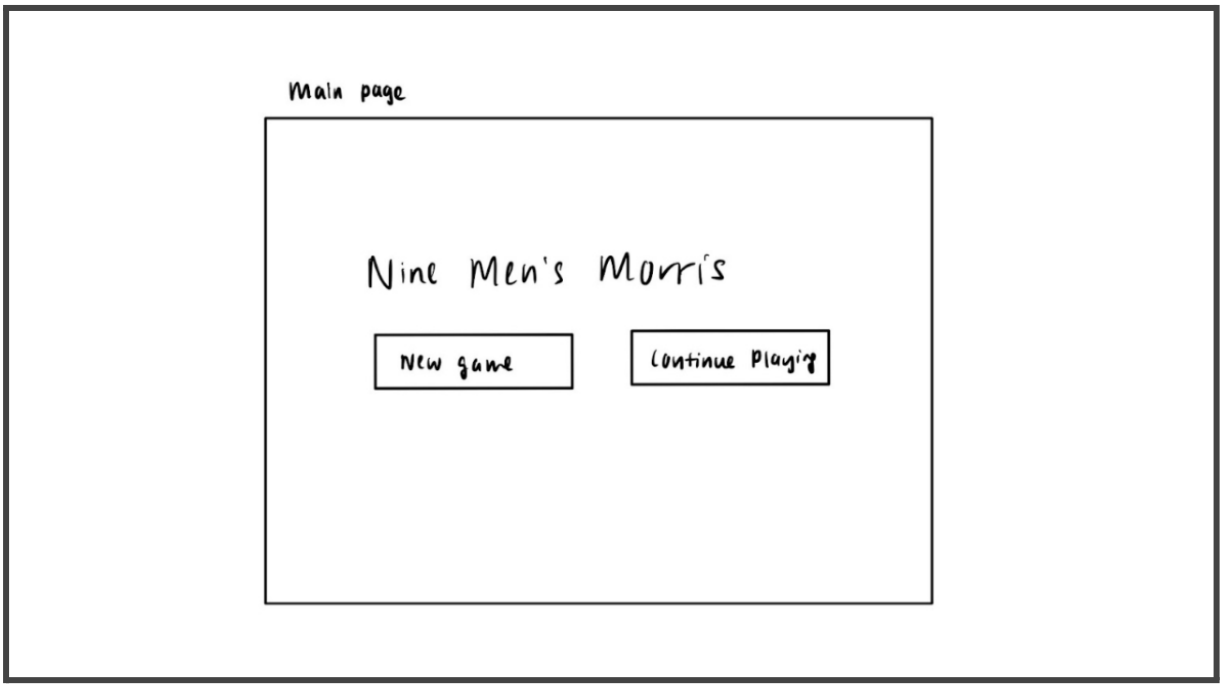


Diagram 1: *The main page.*

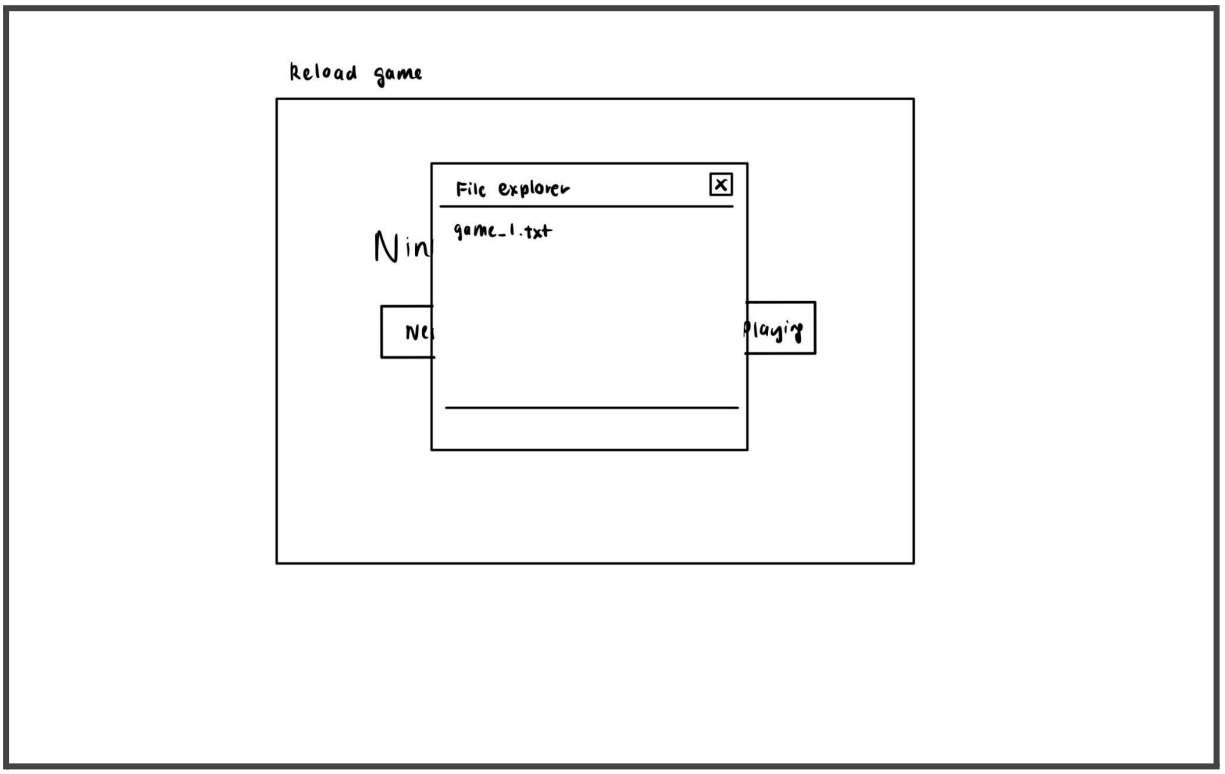


Diagram 2: Players can choose which game to reload to continue playing via File Explorer.

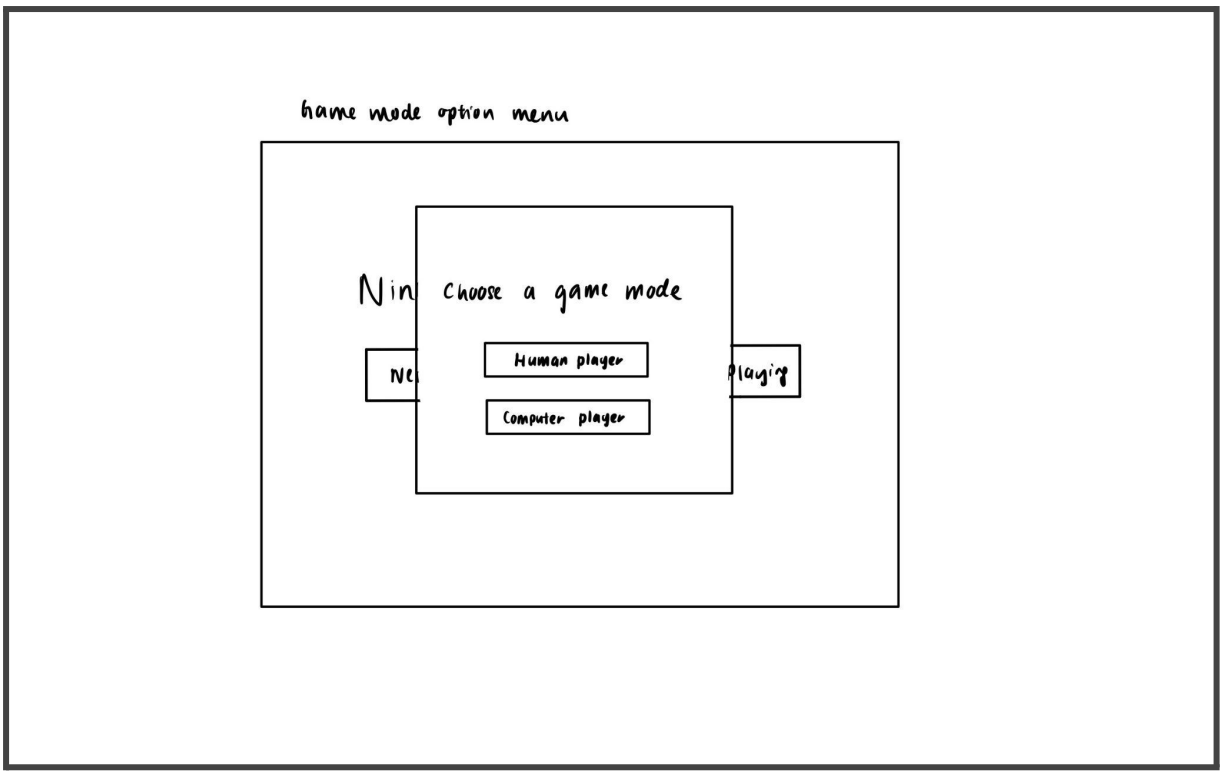


Diagram 3: Players can choose their game mode.

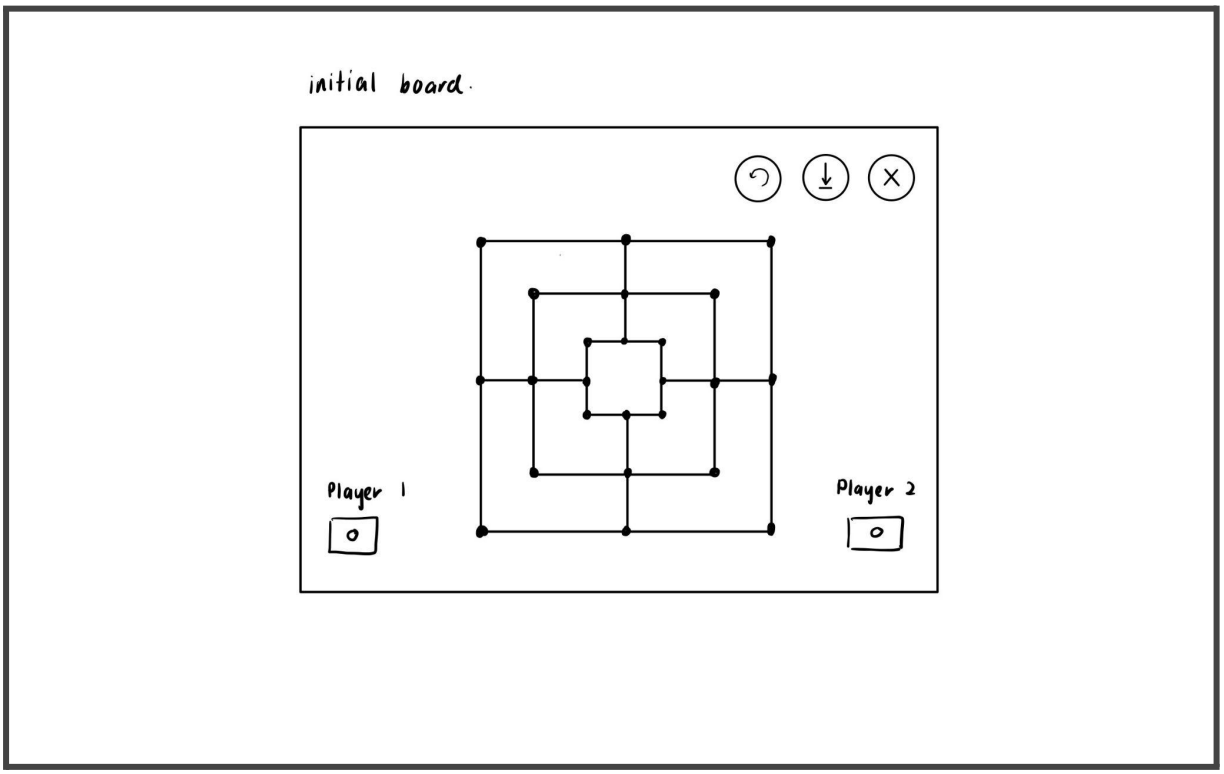


Diagram 4: Initial board position of Nine Men Morris. In the top right corner, from left to right are 3 buttons representing undo, save and exit respectively. The displays in the bottom left and right corner indicate the number of tokens for the respective player that are currently on the board.

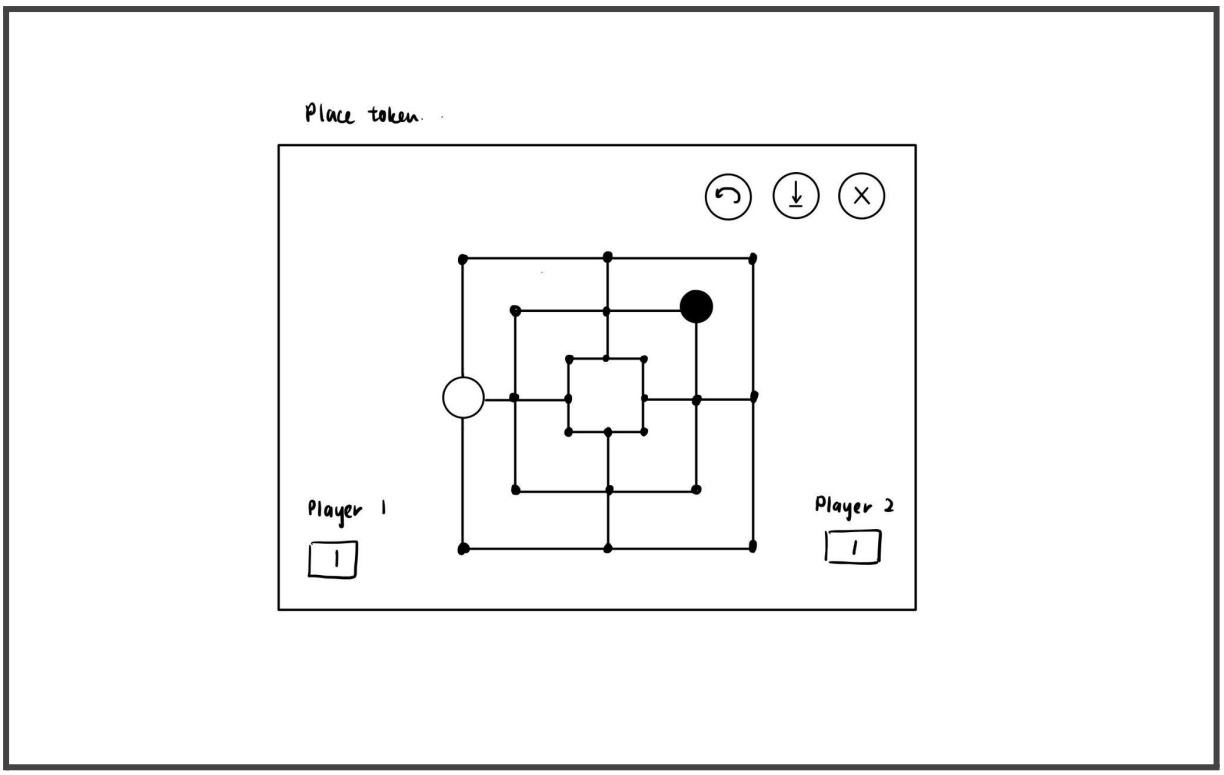


Diagram 5: Players placing their tokens down.

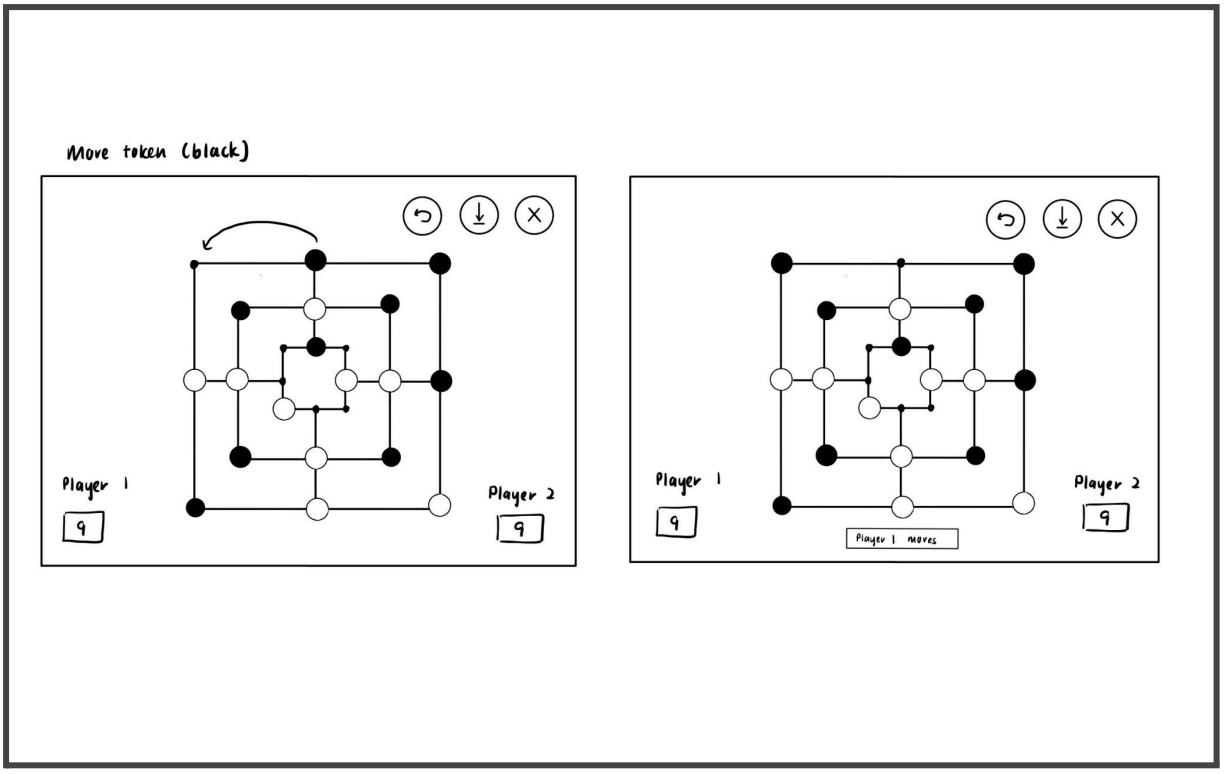


Diagram 6: Player 1 moving his token. Black token is being moved to the left adjacent position.

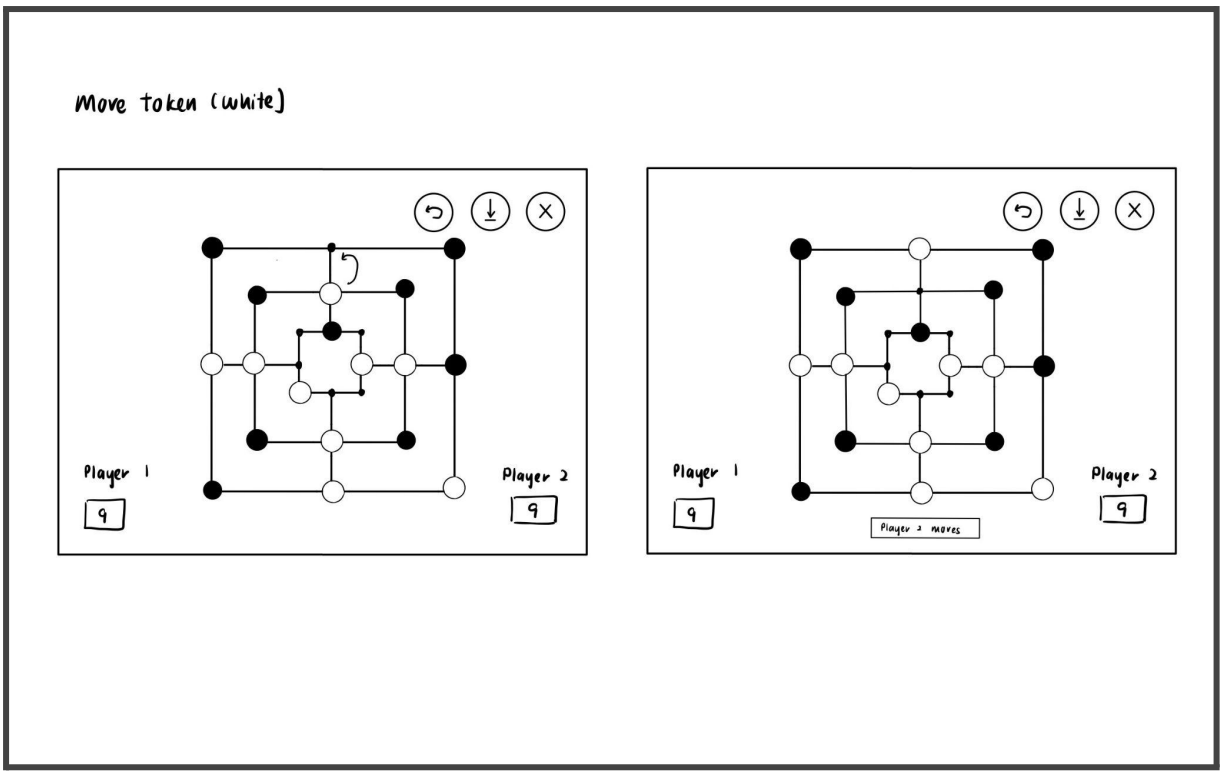


Diagram 7: Player 2 moving his token. White token is being moved up.

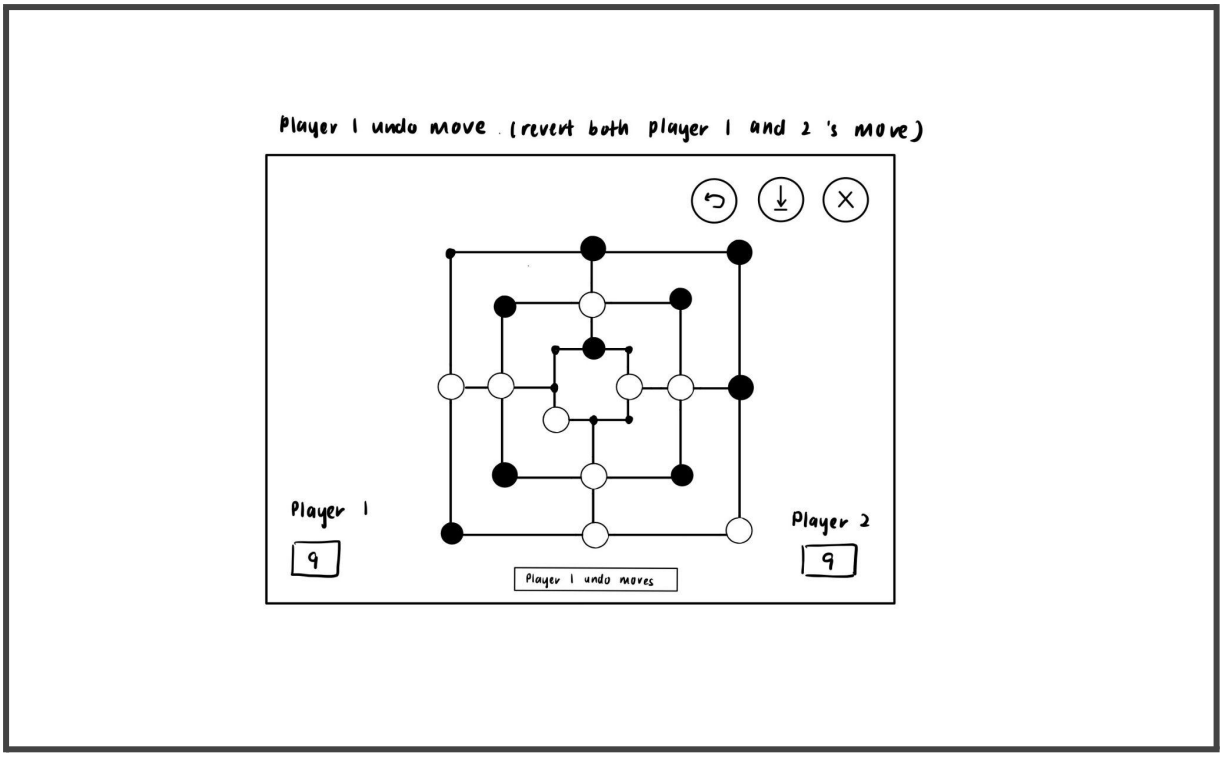


Diagram 8: Player clicks the undo button to undo his move.

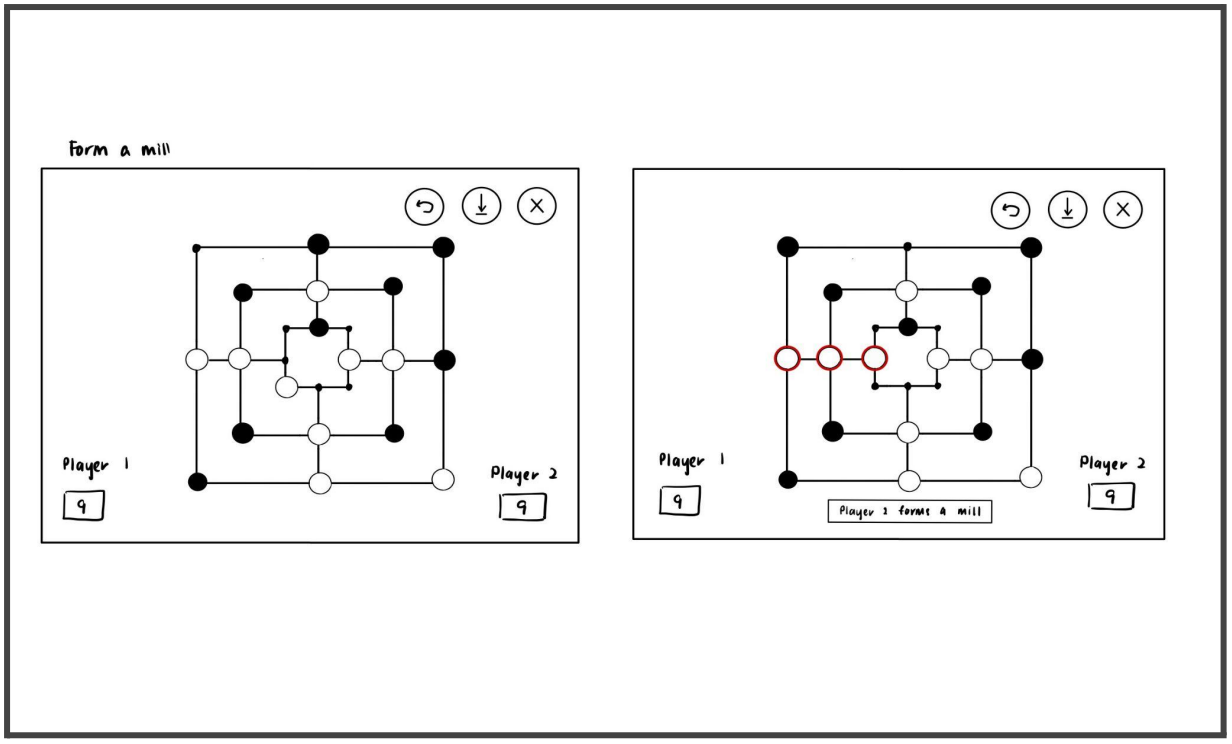


Diagram 9: Player 1 forms a mill.

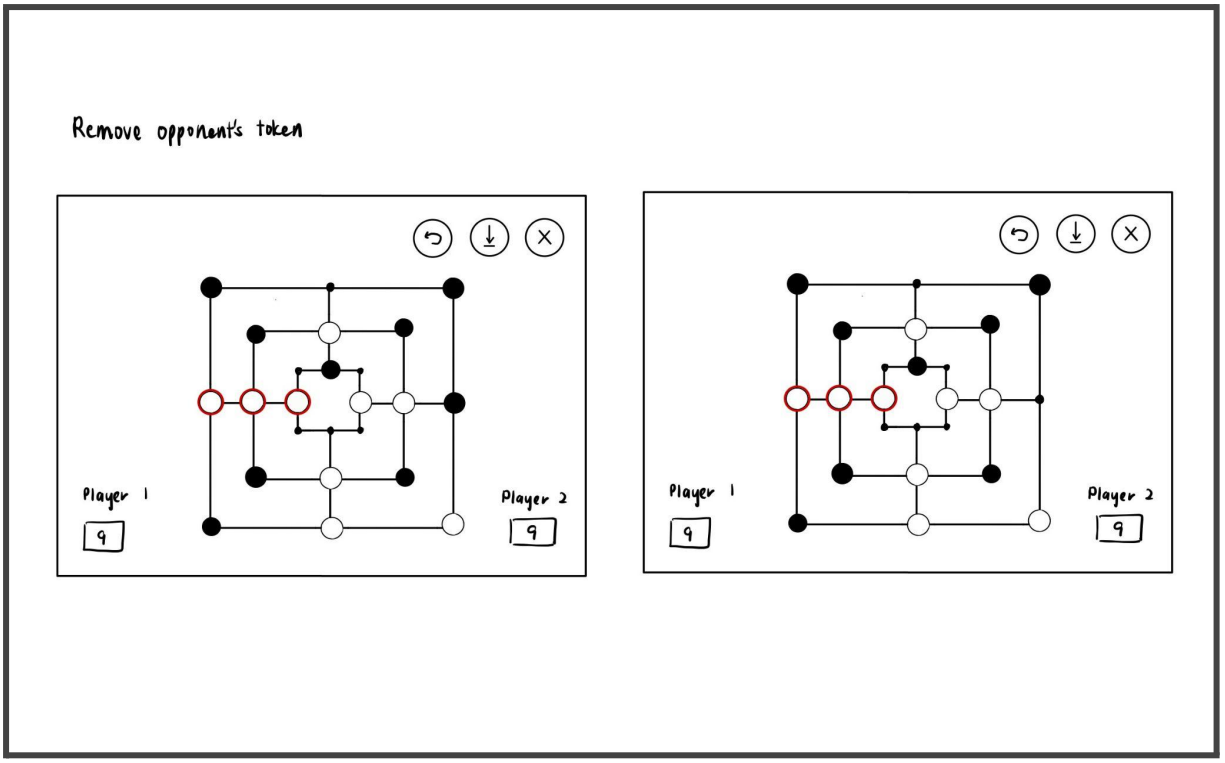


Diagram 10: Player 1 removes a token from Player 2 after forming a mill.

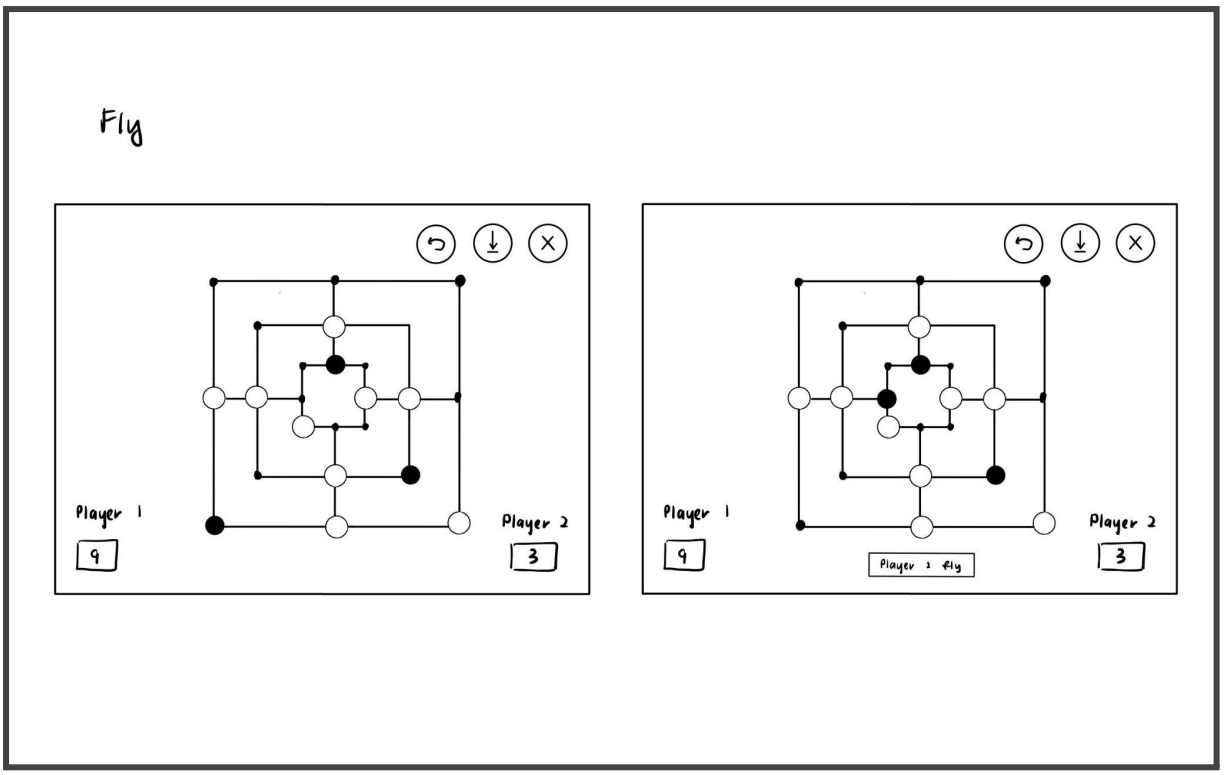


Diagram 11: Player 2 flies to an empty position when he has 3 tokens remaining.

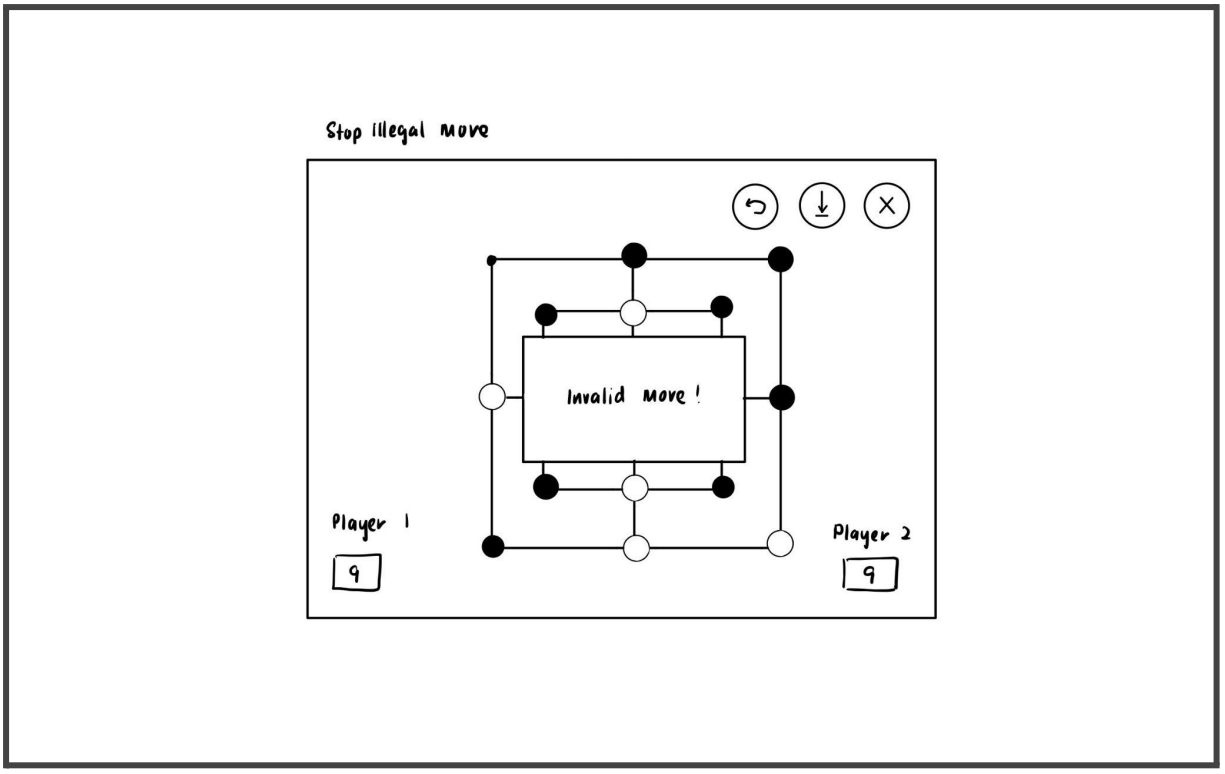


Diagram 12: A pop up message is shown when a player attempts to make an invalid move (i.e. moving the token to an occupied position).

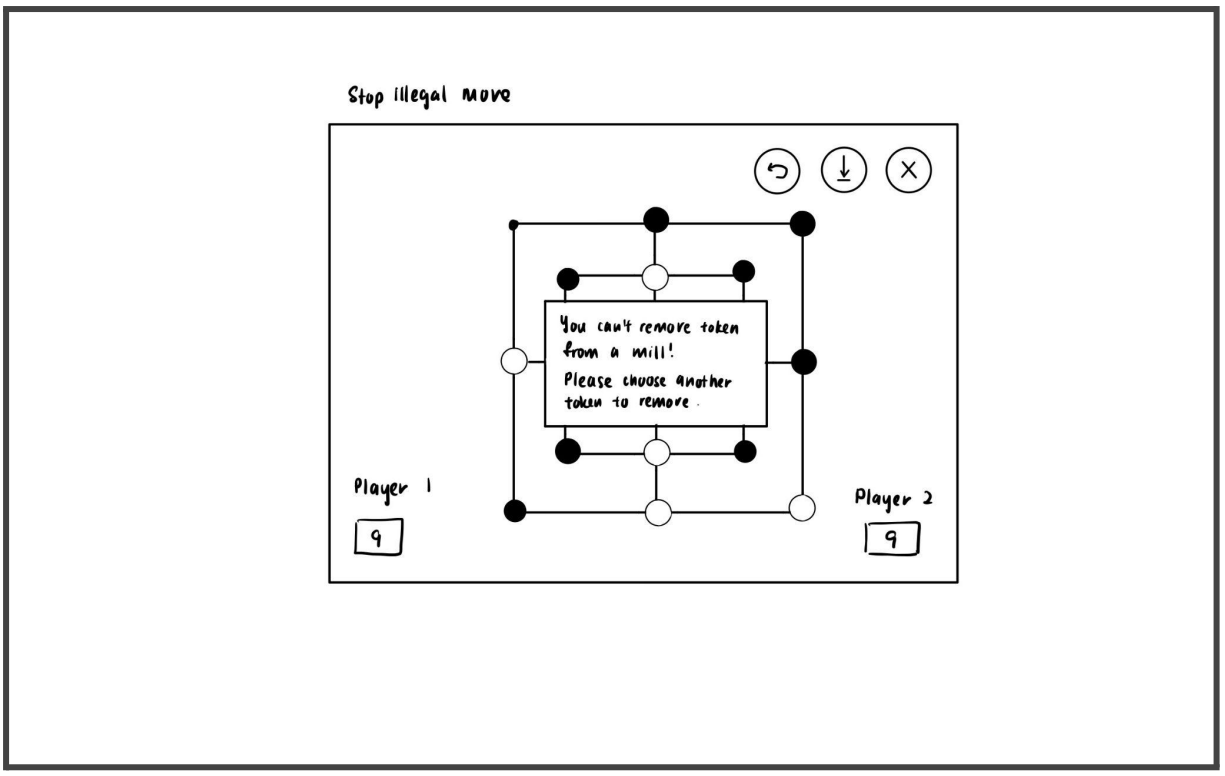


Diagram 13: A pop up message is shown when a player attempts to remove a token from an opponent's mill when there are tokens that are not part of a mill available.

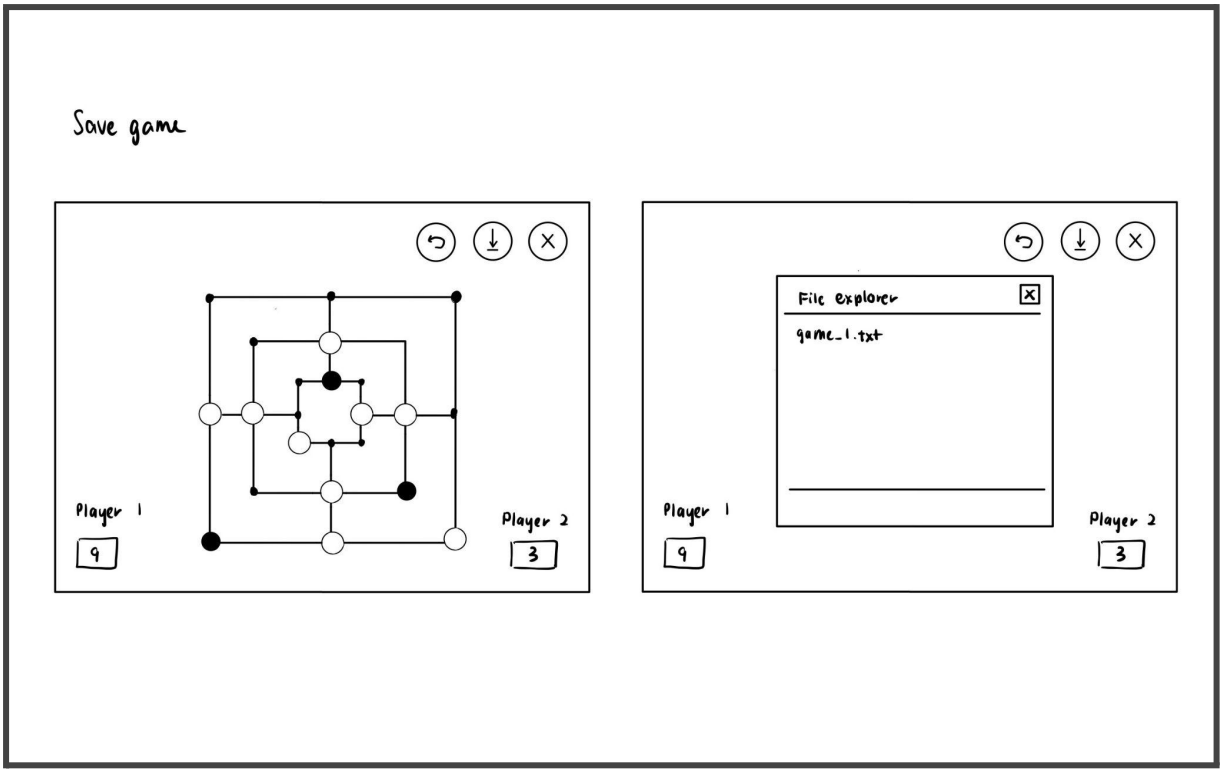


Diagram 14: Player clicks on the save game button to save his game via File Explorer to the player's local device.

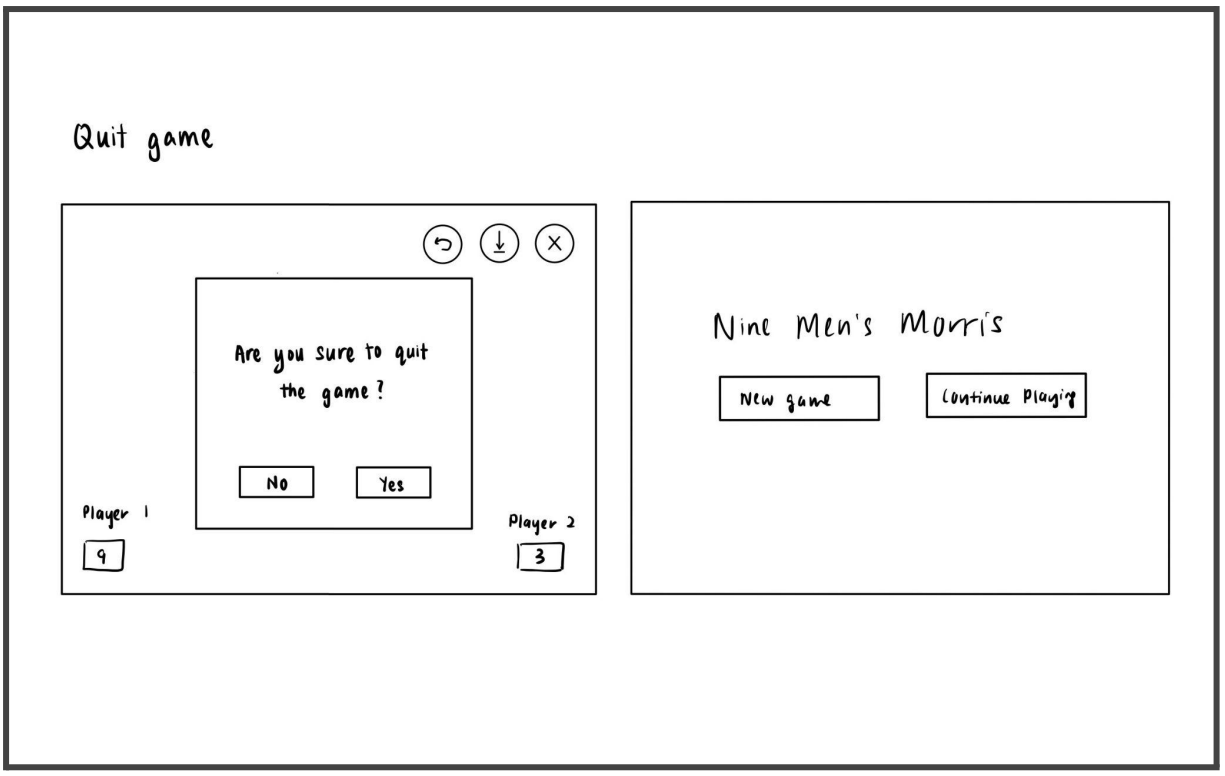


Diagram 15: Player clicks on the exit game button which will bring them back to the main page.

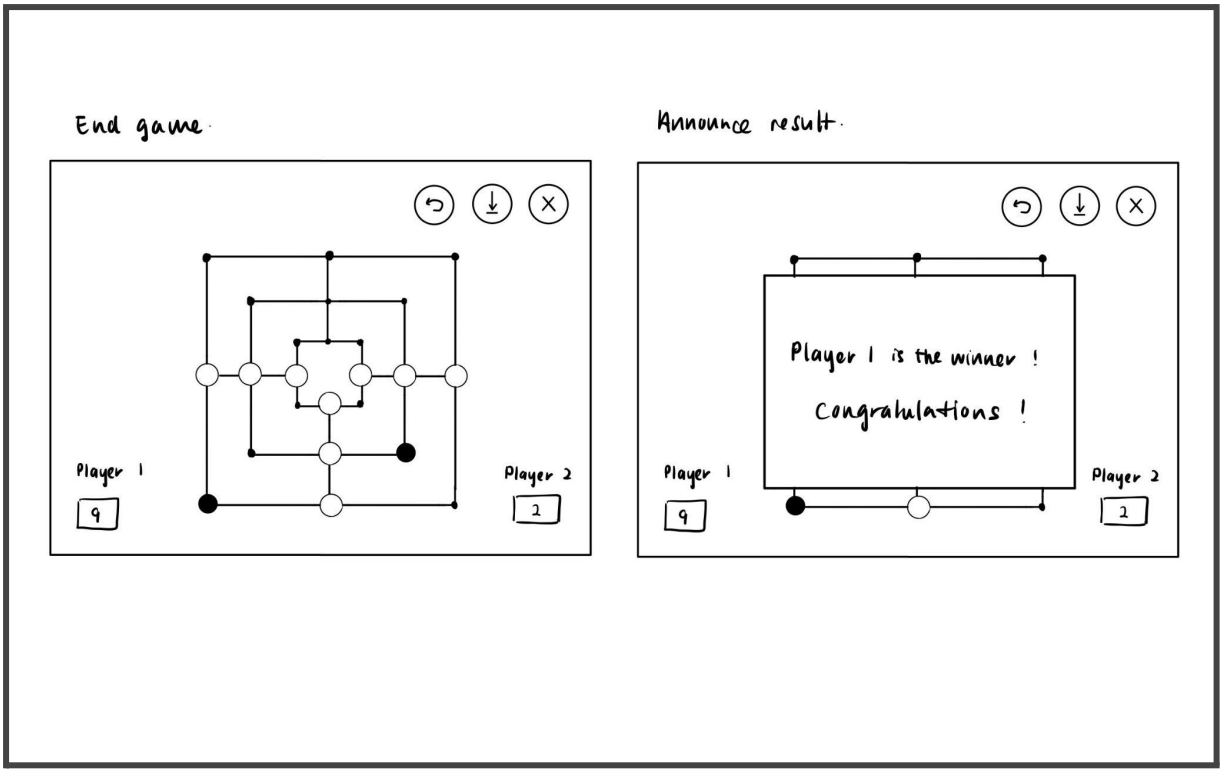


Diagram 16: A win condition has been met (One of the players has only two tokens remaining). The game shows a pop up message to announce the winner and end the game.