FIT3077 Sprint Four 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

Advanced Requirement 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.

Revised 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 be able to **undo** my moves, so that I can perform a different move if I accidentally made a wrong move.
3. As a game player, I want to be able to **save** any unfinished games in a text file so that I can continue playing those games in the future.
4. As a game player, I want to fully **reload** any previously saved game from a text file so that I can resume playing the game.

Game Board

5. As a game board, I want to **save the state** of the current active game in a list at each turn so that the information about the previously made move can be stored and players can undo their moves.

Computer (Opponent)

6. 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.
7. 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.
8. 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.
9. 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.

Our user stories for the selected advanced requirements remain **mostly unchanged** from Sprint 1, **except for user story no. 3, 4, and 5**, the modified part is highlighted in blue. During the previous sprint, we had some confusion regarding the concepts of **saving the game** and **saving the game state**. We initially thought that the game state needed to be saved in a text file. However, after discussing and obtaining clarification, we have decided to implement the saving of the game state and the moves history in an ArrayList structure within our program. This structure will act like a **stack**, allowing us to store and retrieve the latest game state. Additionally, our save game functionality will involve saving the history of game states and the game board into a **text file**. This will enable us to load the saved file and continue playing from where we left off. By addressing this clarification and refining our understanding of the requirements, we are now better prepared to implement the necessary features in a more professional manner.

Architecture and Design Rationale

Architecture for Advanced Requirements

Our first chosen advanced requirement can be divided into 3 parts: (1) player undo moves, (2) player save game, and (3) player reload game. To implement these, we have decided to utilise the **Memento** design pattern due to its convenience of storing objects state without compromising encapsulation. We have introduced several new classes including **GameState**, **GameCaretaker** and **SaveGameAction.** Besides, in order for players to carry out these actions, we have added three new JavaFX Buttons, **Save Game Button**, and **Undo Game Button**. Lastly, we have also added a **Parser** class to assist in reloading games via text files.

By utilizing the GameState class as the Memento, we can capture the state of the Game at different points in time. The GameState class encapsulates all the necessary game information, which is then stored in the history ArrayList within the GameCaretaker. The GameCaretaker is responsible for managing the game's history by storing a list of saved game states and allowing players to **undo their moves by restoring a previous game state**, as shown in Figure 1. Players can initiate the undo action by clicking the Undo Game Button. Distributing the responsibilities among these classes follows the **Single Responsibility Principle (SRP)** and allows for a clear separation of concerns. The GameState class focuses on representing a specific state of the Game, the Game class handles the game logic and state management, and the GameCaretaker class takes care of storing and retrieving the game states for save, undo and reload operations. This division of responsibilities promotes encapsulation and flexibility in managing game states.

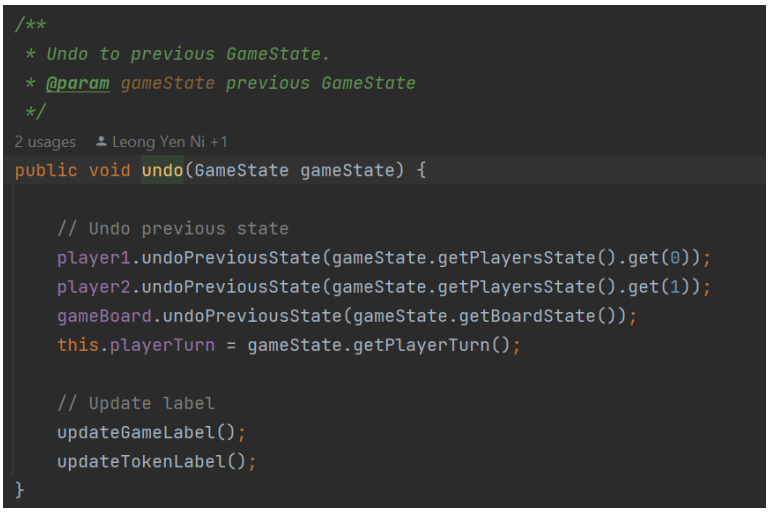


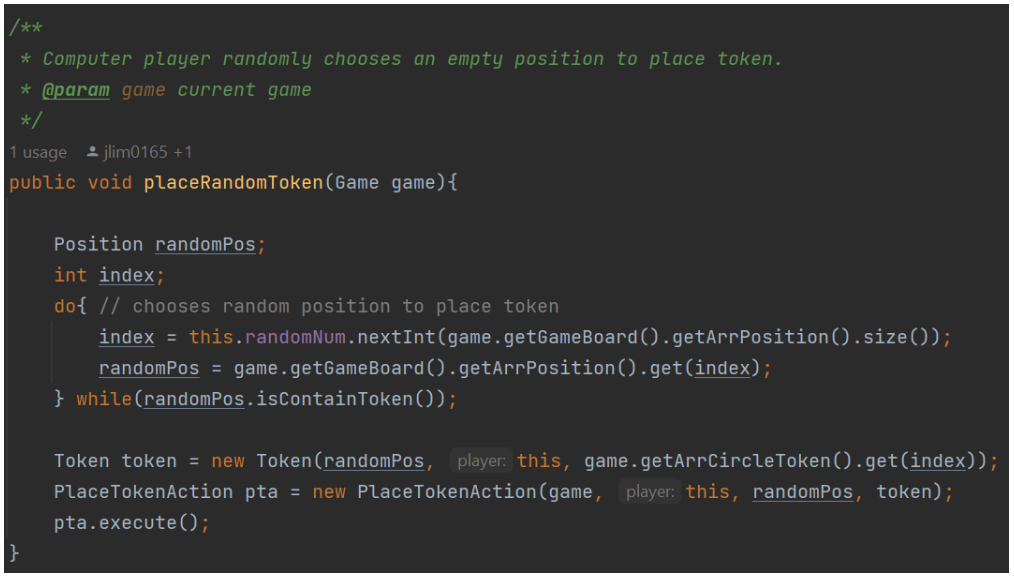
Figure 1. Method (in Game class) that restores previous game state. This function is called in GameCaretaker.undoState().

The game is automatically saved at the end of each game turn and after forming a mill by creating a new GameState instance. The Save Game Button will execute the SaveGameAction to enable players to select a text file (.txt) through the file explorer. The selected file is then used to **save the entire game history**. Essentially, the game history is written to a text file, allowing players to later reload the game by choosing the respective file.

To reload the game, a player selects a text file from file explorer. The game reads the file, and converts the history back into gamestate to be stored under the GameCaretaker. Lastly, the game restores the latest game state from GameCaretaker's history. Note that the restoration of the latest game state **reuses the code from the undo action** in Figure 1, thus we abide by the **DRY Principle**.

For the second advanced requirement, we have introduced a **ComputerPlayer** class that inherits from the Player class. This class includes three main methods that correspond to the token actions: placing a token, moving a token, and removing an opponent's token, one example is illustrated in Figure 2. Each of these methods randomly selects a token or position to perform the action. In terms of maintainability, our code's design and structure have facilitated the implementation of this advanced feature. By utilizing inheritance, we were able to easily extend the existing Player class to create the ComputerPlayer class. Additionally, our code's maintainability is evident in the separation of concerns. The Player class focuses on the general functionality and attributes of a player, while the ComputerPlayer class specifically handles the random moves behaviour. This clear separation allows for easier maintenance and future enhancements, as modifications or improvements to the random moves behaviour can be made within the ComputerPlayer class without affecting the core Player class or other parts of the codebase. Furthermore, we have implemented a delay in the execution of the computer player's actions to improve the user experience. This delay was achieved using JavaFX's **Timeline** and **KeyFrame**, providing a smoother animation for token

movements. This enhancement not only enhances the visual appeal of the game but also makes the gameplay more engaging for human players.

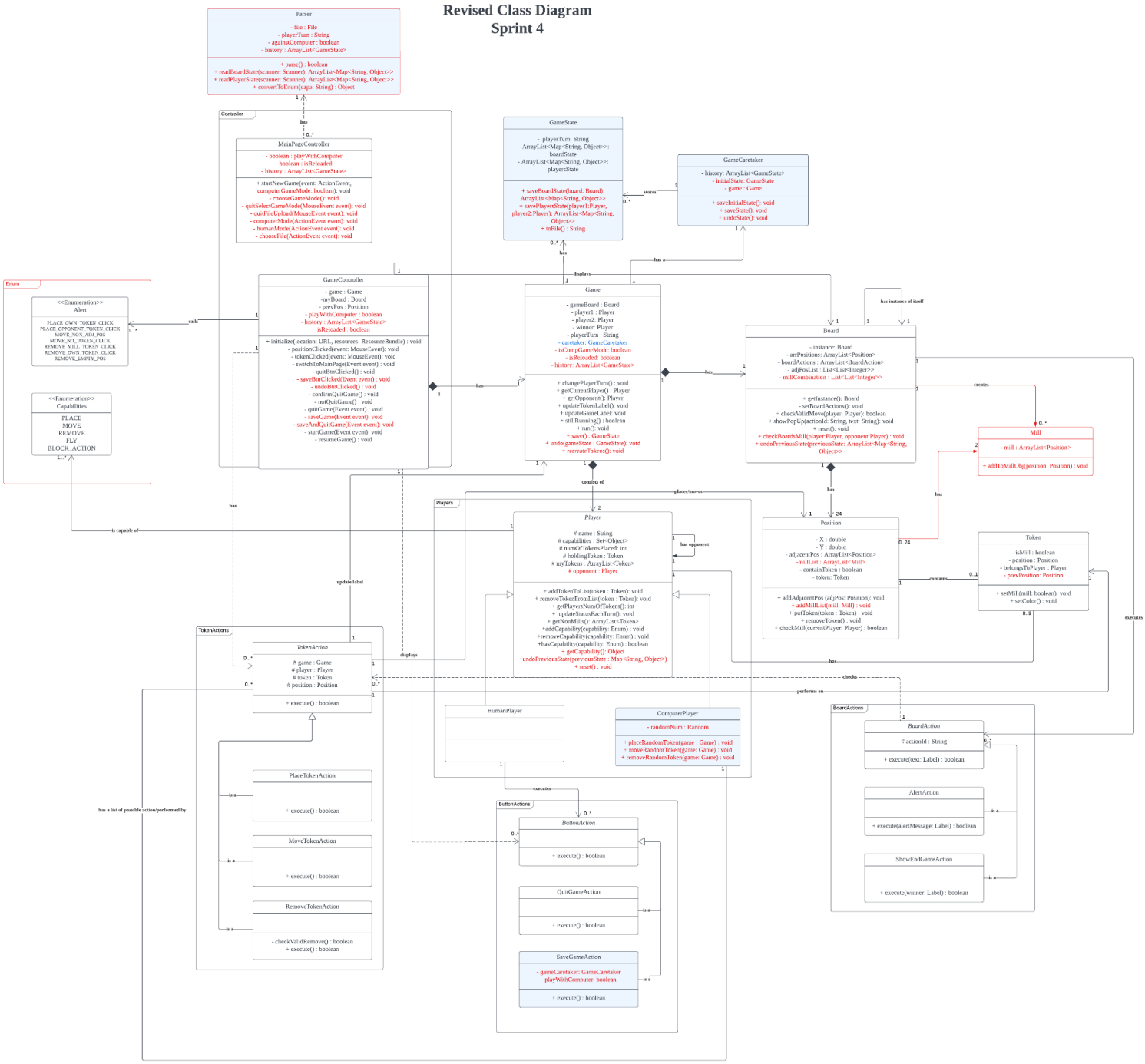


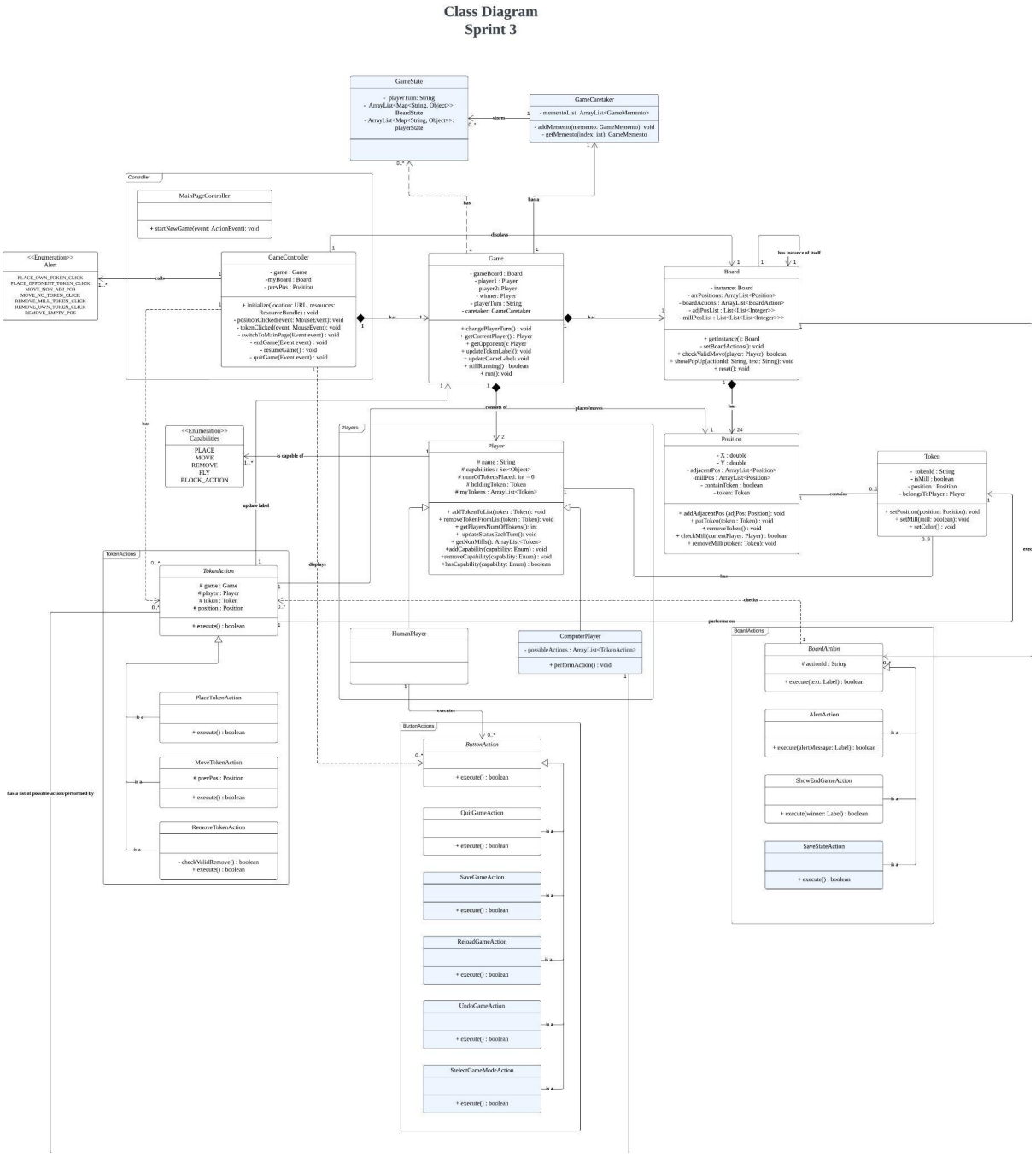
```java
/**
 * Computer player randomly chooses an empty position to place token.
 * @param game current game
 */
1 usage  jlim0165 +1
public void placeRandomToken(Game game){

    Position randomPos;
    int index;
    do{ // chooses random position to place token
        index = this.randomNum.nextInt(game.getGameBoard().getArrPosition().size());
        randomPos = game.getGameBoard().getArrPosition().get(index);
    } while(randomPos.isContainToken());

    Token token = new Token(randomPos,  player: this, game.getArrCircleToken().get(index));
    PlaceTokenAction pta = new PlaceTokenAction(game,  player: this, randomPos, token);
    pta.execute();
}
```

Figure 2. Method for computer to randomly place a token.

Justification on Revised Class Diagram

Link:
https://lucid.app/lucidchart/e66943cd-742a-4ce3-8dd3-ba4812ec3b4b/edit?viewport_loc=1561%2C76%2C5288%2C2997%2CTpqZ9W7NyXUI&invitationId=inv_2b133f71-657b-4f23-99c3-87466c12043d





(For comparison purposes)

In the updated class diagram, the elements highlighted in red indicate the new additions or revisions made during the current sprint. One significant change we made was the introduction of a dedicated class **Mill**, for mills. In the previous prototype, we relied on a double nested list with hard coded integer values to initialise the mill positions for each position. However, we recognized the need for a more object-oriented approach and decided to create a separate class specifically for handling mills. This improvement aligns with the principles of object-oriented programming and allows for a more flexible and maintainable representation of mills in the game.

Another architecture design we created is implementing a Parser class to read and convert the text files back into game states. This is to separate the logic away from the class where the "Load Game" button is located, which follows the **Single Responsibility Principle (SRP)**. The Parser class also makes it easier to modify the code to read different types of files in the future.

Furthermore, for implementing a computer player in our game, our initial plan was to use a List to store the possible actions or moves that computer players can make in different game phases. However, as we progressed with implementing this advanced requirement, we recognized that this approach is not practical for the Nine Men Morris Game. This is primarily because the game involves only three distinct token movements, which are place, move/fly and remove, executed in different phases of the game. With that, we made the decision to simplify the implementation by using individual methods for each action. Within these methods, random tokens and positions are selected to enable the computer player to make a valid move. This streamlined approach proved to be more effective and efficient for the specific requirements of the game, ensuring the computer player can perform appropriate actions without the need for a comprehensive list of all possible moves.

Justification on Implementation of Advanced Requirements

The chosen advanced features for our game were finalised during Sprint 1 and remained the same throughout the project, as we had no reasons to change them.

The implementation of the first advanced feature, which involves saving, undoing, and reloading the game, was slightly challenging but still manageable. By leveraging the good design practices employed in previous sprints, such as the use of abstract classes and inheritance, our codebase remained well-structured. This allowed us to seamlessly introduce new classes without the need for extensive rewrites. For instance, SaveGameAction, extended the existing ButtonAction, showcasing the benefits of inheritance and obeying **Open-Closed Principle (OCP)**. Furthermore, in a previous sprint, we decided to utilize the Memento design pattern to implement this feature. As part of this approach, we introduced two key classes, GameState (formerly known as GameMemento) and GameCaretaker classes. This Memento design pattern offered a convenient way to incorporate undo, save, and reload functionalities, and these newly introduced classes were designed with clear responsibilities, as described in the Architecture for Advanced Requirements section. This approach enables code reuse and maintainability, which enhances our quality of implementation. The slightly tricky part was deciding at which point in time the game should be saved, the data to be saved, and how to restore a previous state based on the data saved.

The design of our text file follows a CSV style format. While we were debating between a CSV or JSON style format, we decided to go with CSV because CSV is more suitable for our use case. JSON is advantageous if we had a large variety of data, but our data is nice and structured so there is no need for it. While both formats are considered lightweight, CSV is so

simple to parse that Java's built-in Scanner class can parse it, whereas JSON typically requires external libraries such as GSON to parse it. Ultimately, since the data we are storing is so simple, CSV is more suitable. Even if we wanted to store more information in the future, we can just add more columns. Lastly, we have to save the entire history of the game so that players can undo their moves even on reloaded games.

```
HUMAN
2
0|true|false|Player 1
1|false|null|null
2|false|null|null
3|false|null|null
4|false|null|null
5|false|null|null
6|false|null|null
7|false|null|null
8|false|null|null
9|false|null|null
10|false|null|null
11|false|null|null
12|false|null|null
13|false|null|null
14|false|null|null
15|false|null|null
16|false|null|null
17|false|null|null
18|false|null|null
19|false|null|null
20|false|null|null
21|false|null|null
22|false|null|null
23|false|null|null
Player 1|1|PLACE
Player 2|0|PLACE
```

Figure 3. Sample snippet of saved data. HUMAN is the document header which tells us the game mode. 2 is the player whose turn it is to move. The next 24 lines represent the status of each position, which are the index, if it contains a token, if it is a Mill and the player the token belongs to. The last 2 lines are information about the players, which are player name, number of tokens placed and capabilities. This is just one game state, each game state is separated by a blank line.

Based on the advanced requirement, "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". We have deviated slightly from this approach. While the approach mentioned above has the advantage of storing minimal data, restoring the latest version of the game would require replaying the entire game. Let n be the number of iterations of the game played. Thus, restoring the game would require O(n) time complexity. In contrast, our approach stores more data, but we only need to restore the latest game state regardless of how many iterations of the game has been played. This gives us a time complexity of O(1). Lastly, we have chosen to trade off a bit of memory space for extra readability. Since we are already using text files which are considered to be lightweight, and we do not have a compelling reason to minimize the size of our saved file, we decided that

saving some extra information at the expense of memory will prove valuable when we need to read the files for debugging purposes.

Next, for the second requirement, implementing the computer player functionality was relatively straightforward compared to our first advanced requirement. As mentioned earlier, we utilised distinct methods within the ComputerPlayer class to handle valid token movements. Since we just reused the TokenAction classes implemented in previous sprints, the overall implementation of a computer player was manageable.

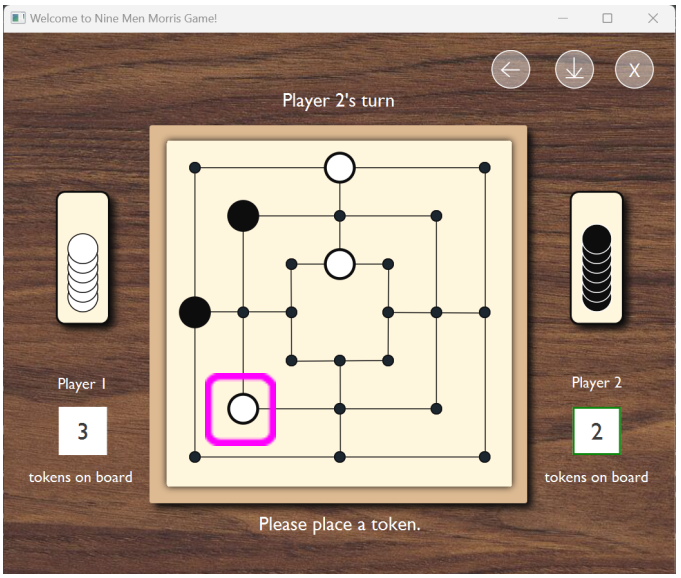
Demonstration of Game

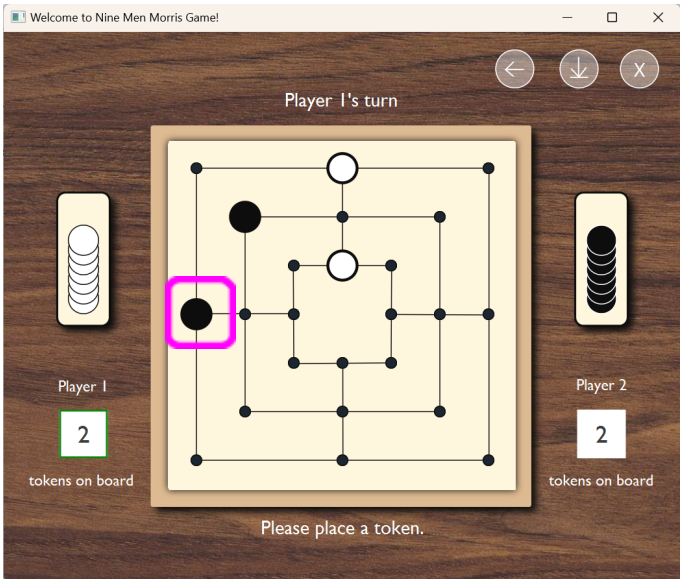
Link to video demonstration: https://youtu.be/ycoocf1rIY8

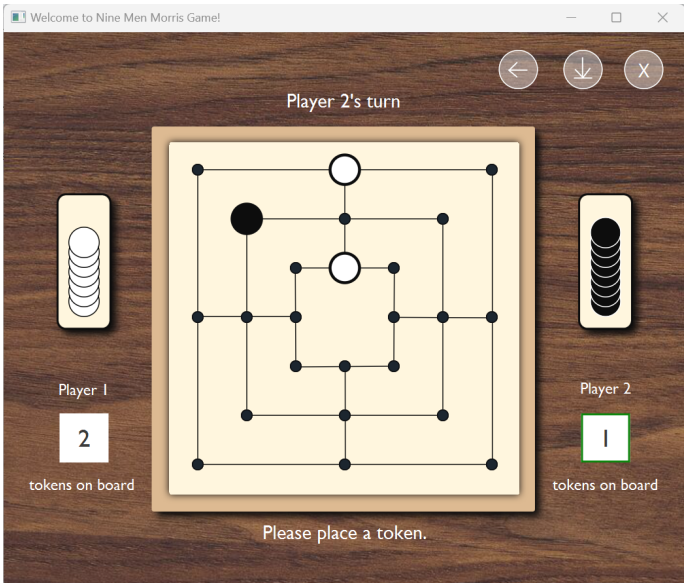
Undo moves

a) Playing with human players

1. Undo placing token

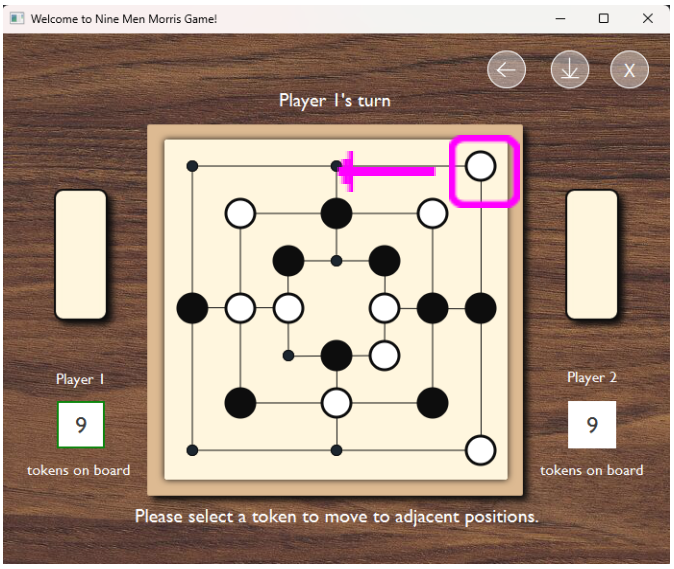


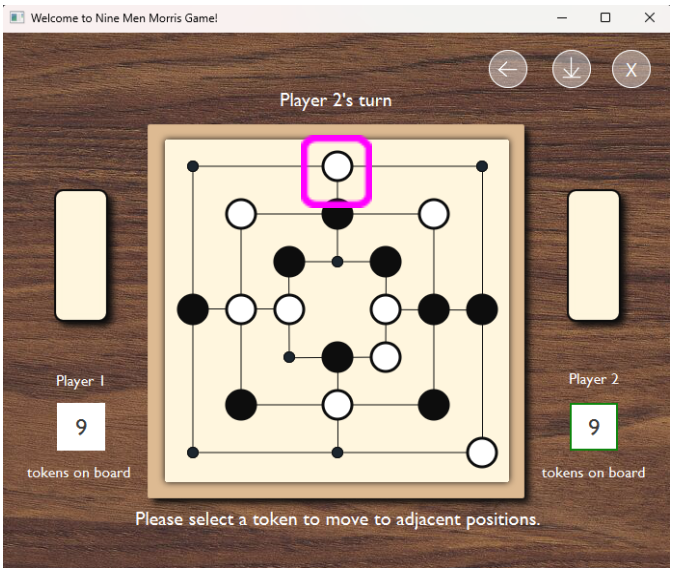




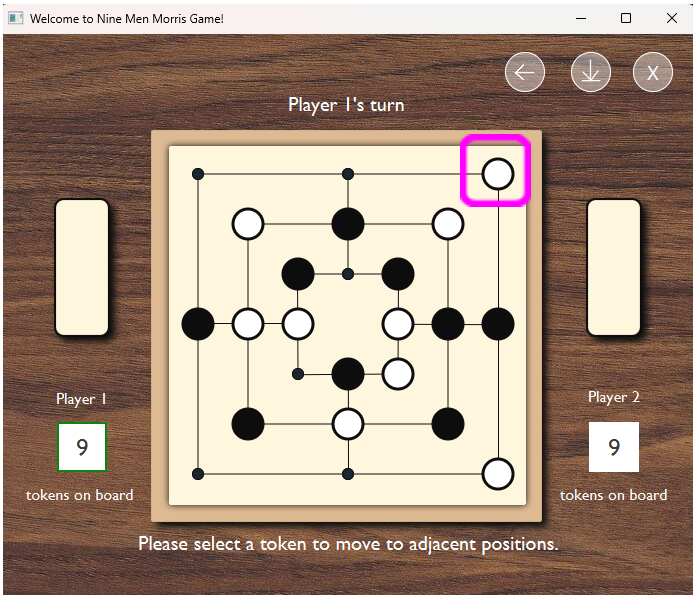
In this example, the undo button is clicked twice, every once the undo button is clicked, the move made by the previous player will be withdrawn, and the previous player is required to make its move again (place the token).

2. Undo moving token

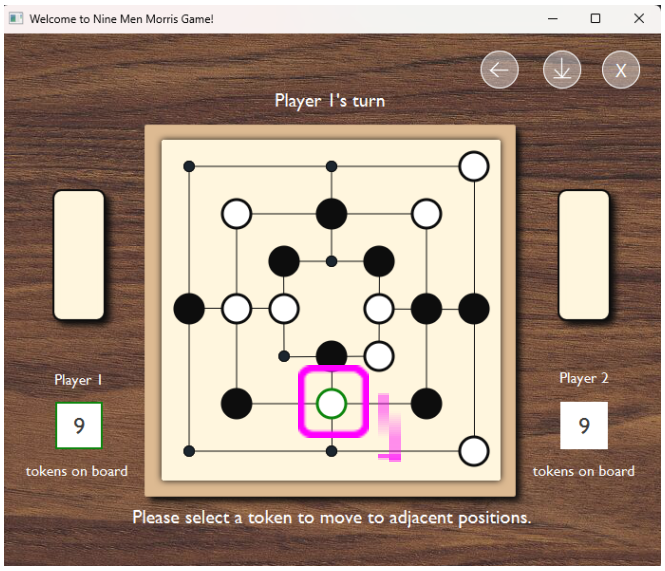


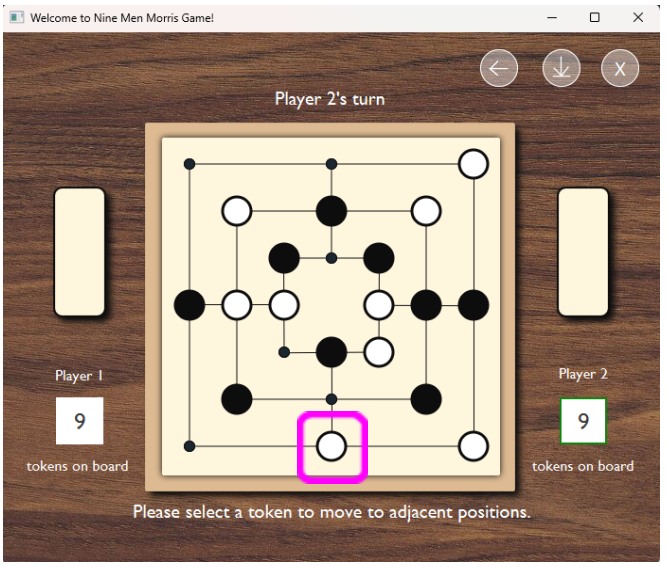


Player 1 selects a token and makes a move to the left.



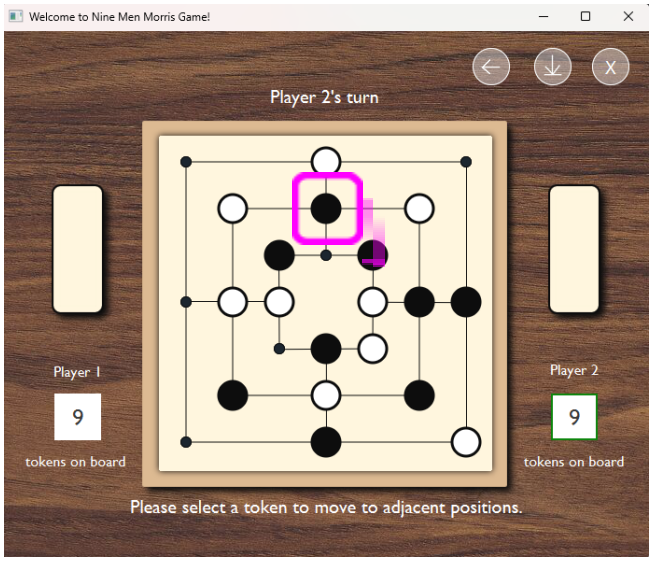
Then, Player 1 clicks the undo button and the move is withdrawn. The Board is reverted to the state before Player 1 moves the token. Player 1 is able to make a move again.

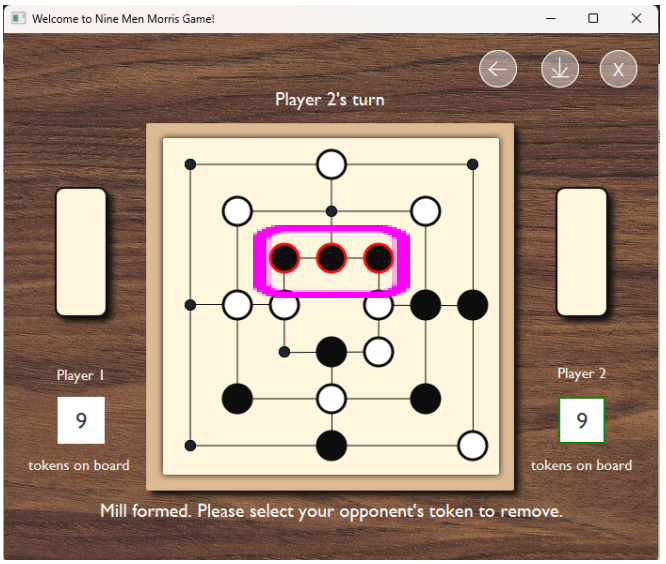




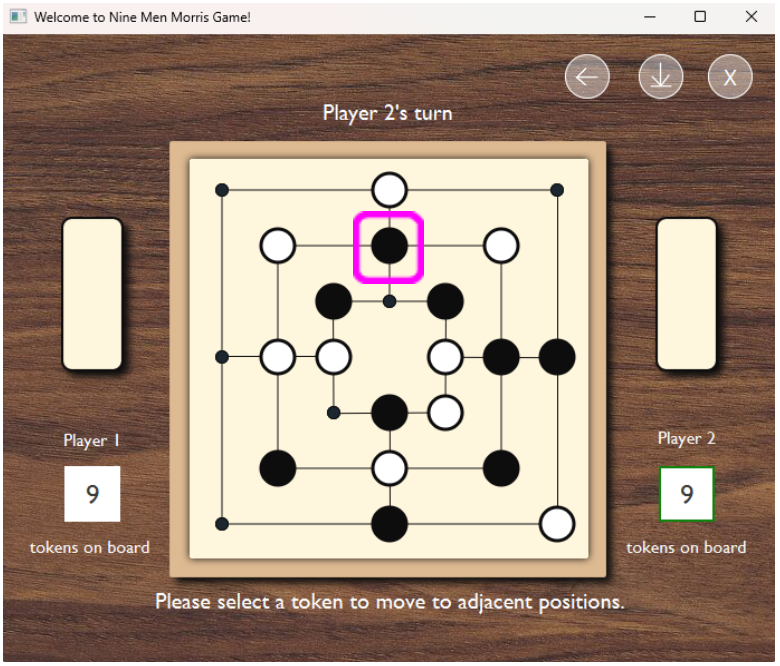
Player 1 selects another token to move.

3. Undo forming mill



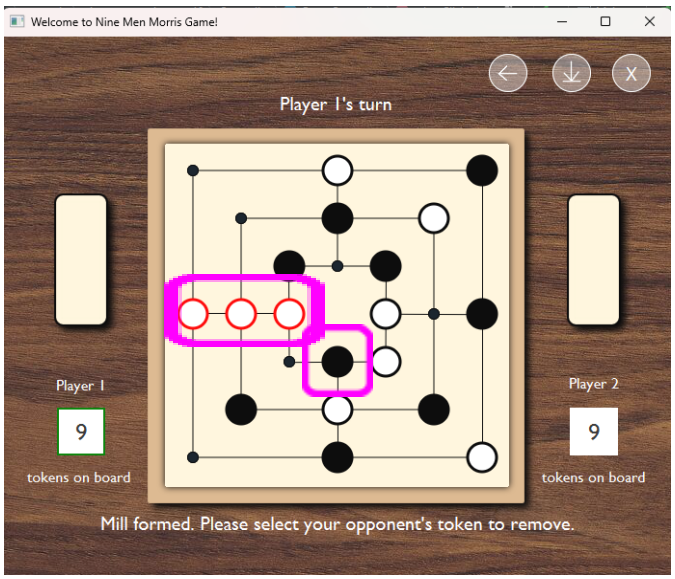


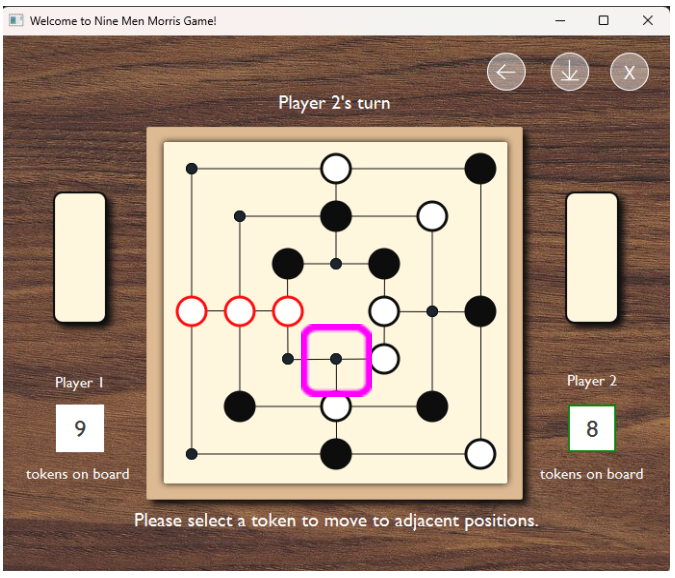
Player 2 moves a token and forms a mill.



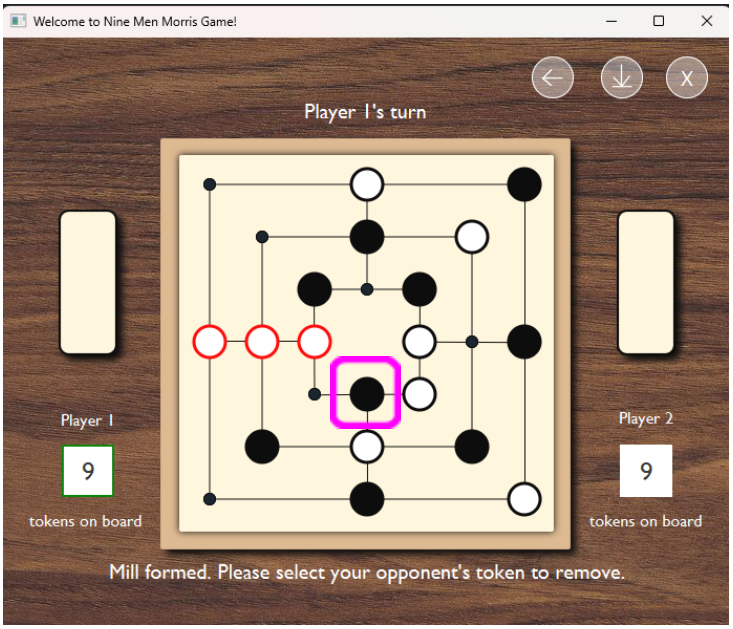
Player 2 undo his move. The game is reverted to the state before forming the mill. Player 2 can make a move again.

4. Undo removing token

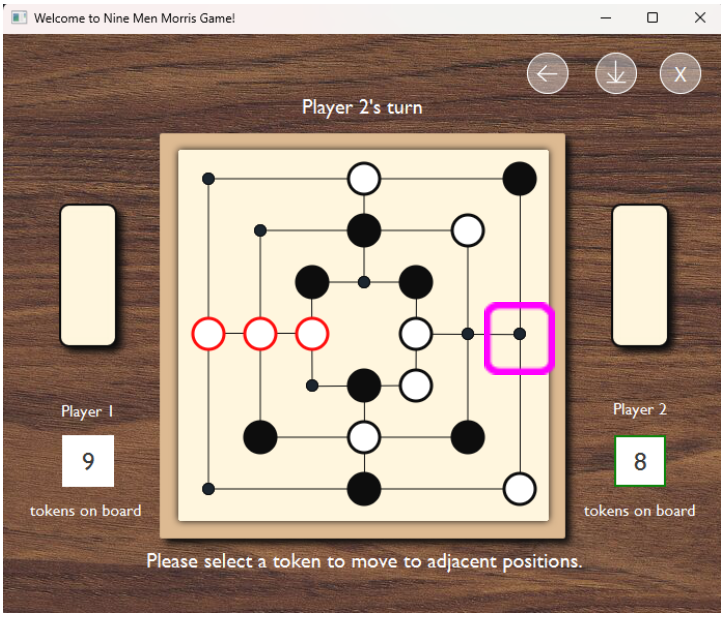




Player 1 forms a mill and removes Player 2's token.



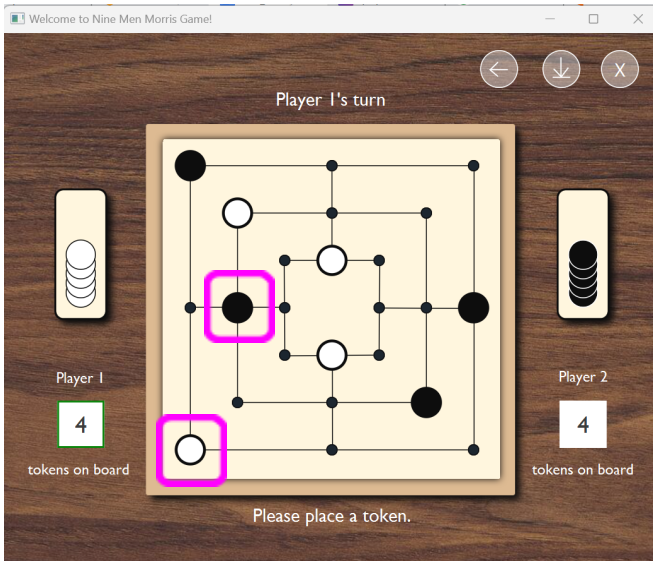
Player 1 undo his move. The game is reverted to the state before Player 1 removes the token. Player 1 can reselect another token to remove.

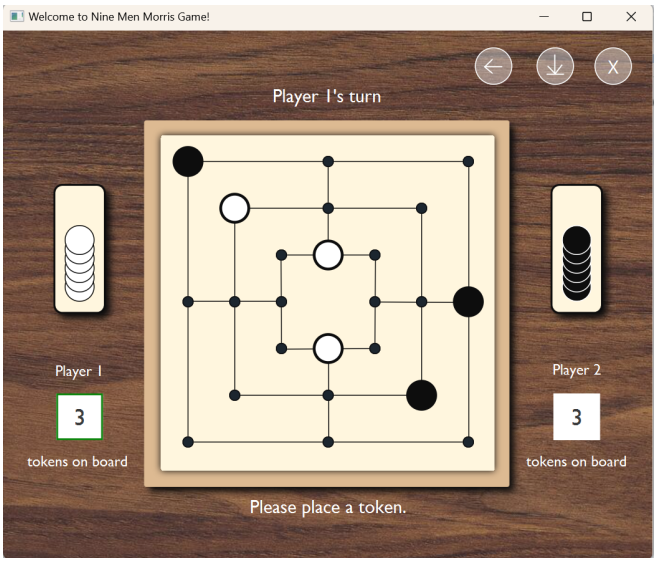


Another token is removed from the board.

b) Playing with computer

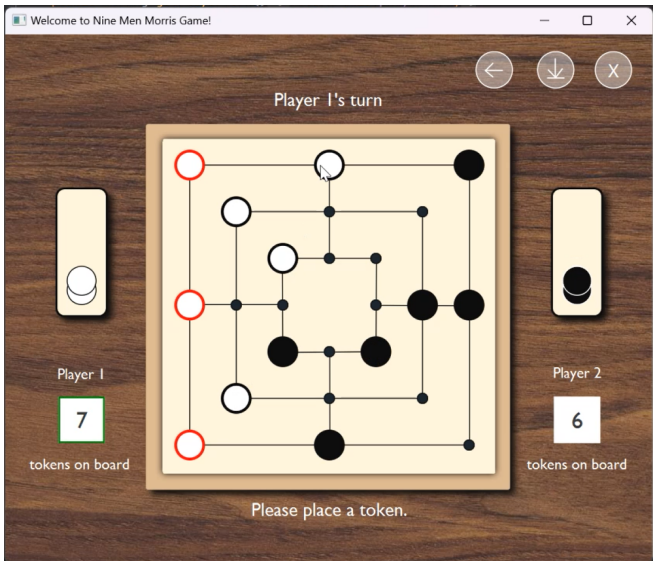
1. Undo placing token

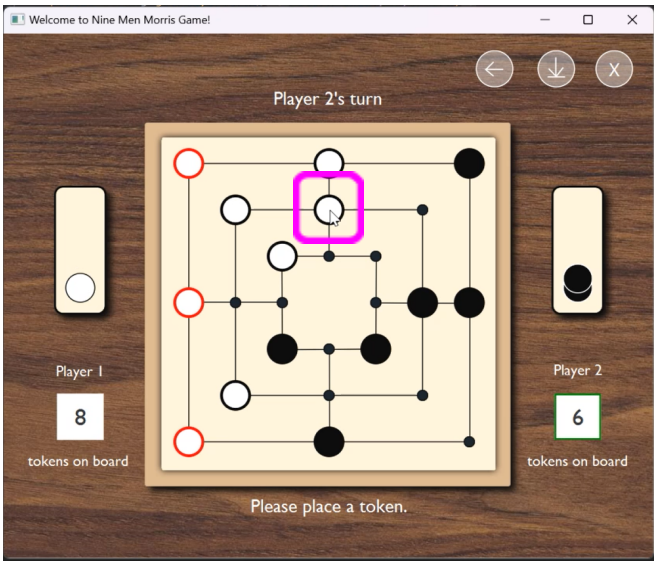


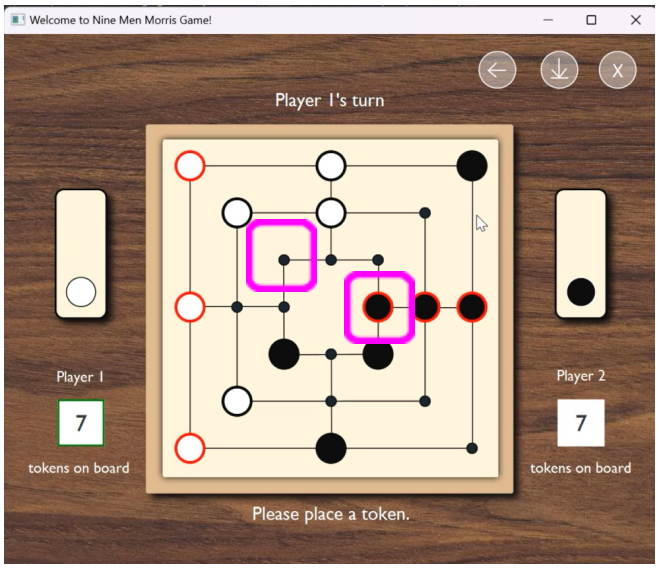


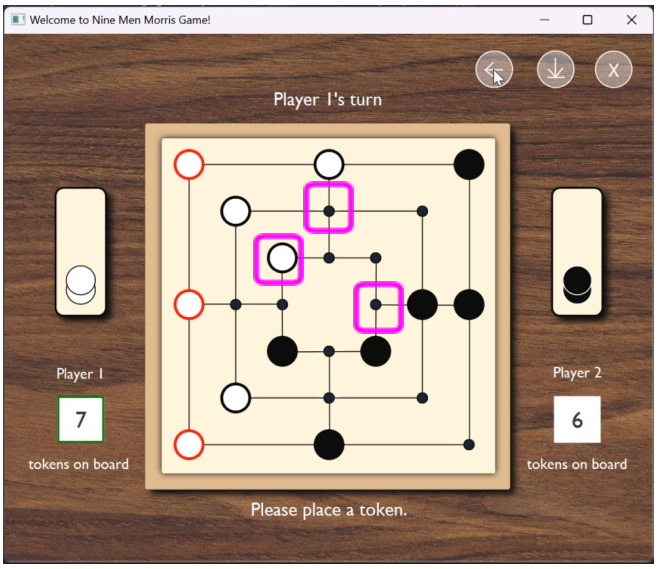
When playing with a computer, every once the undo button is clicked, the computer's move along with the previous human player's moves are undone. Human player is required to make its move again, then the computer will make its move as well.

2. Undo when computer forms mill





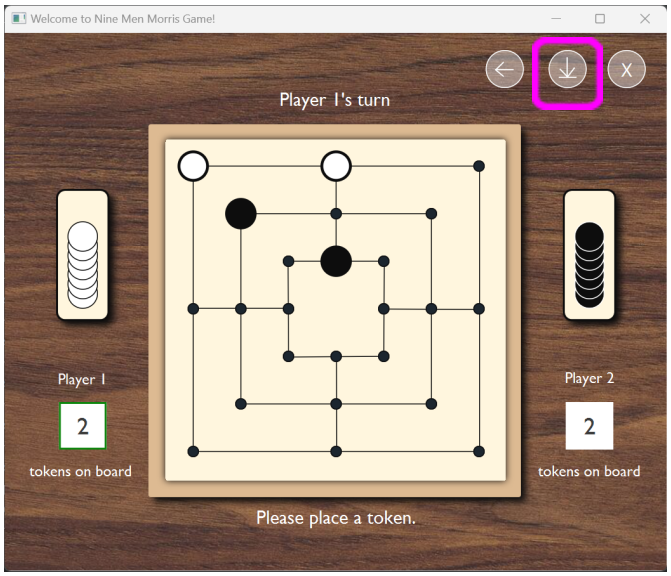


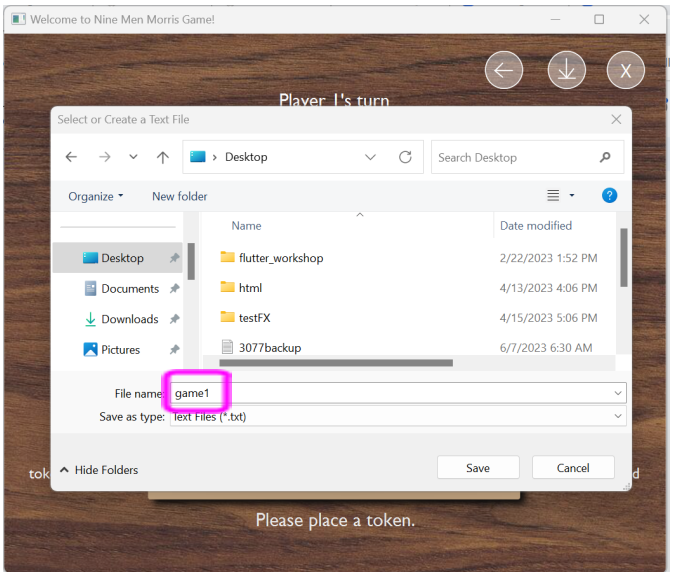


When computer forms a mill and removed human player's token, now it is human player's turn, and if human player clicks undo, we will undo the computer's remove token action, move/place token action, and human player's latest performed action so that player can re-perform the action after undo. As demonstrated in the pictures above, the first and second picture shows that human player placed token. The third picture shows that computer placed token and formed mill. The last picture shows after human player clicked on undo, the game will return to picture 1, which computer's place and remove token action, as well as human player's last place token action are undone, and player 1 can now perform place token action again on any position.

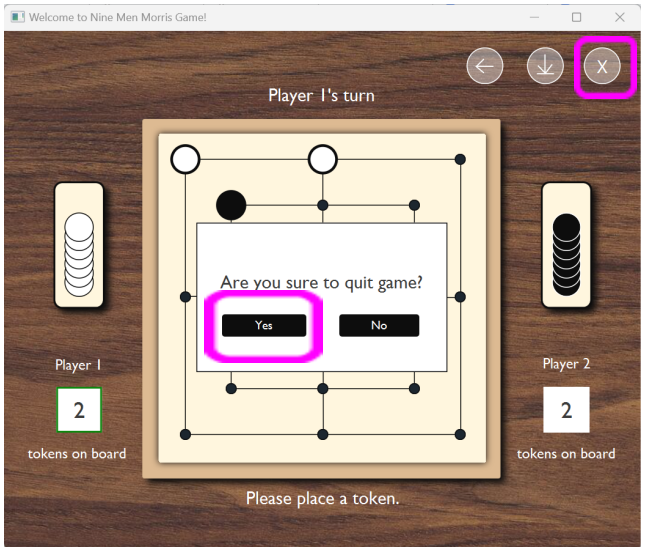
Save and reload game

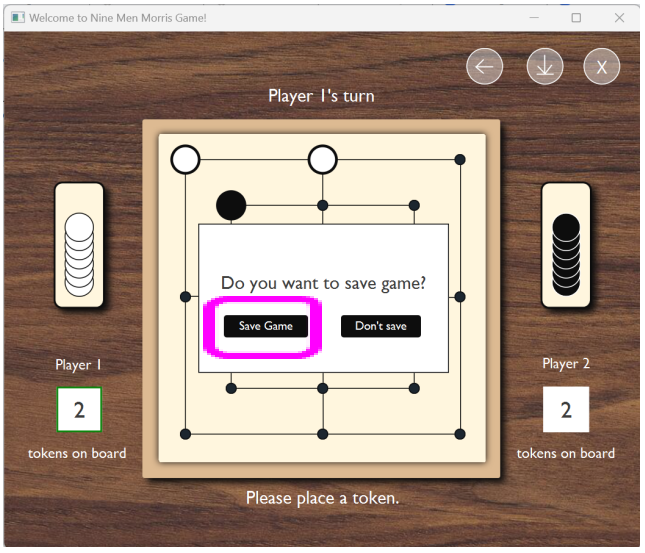
a) Save game





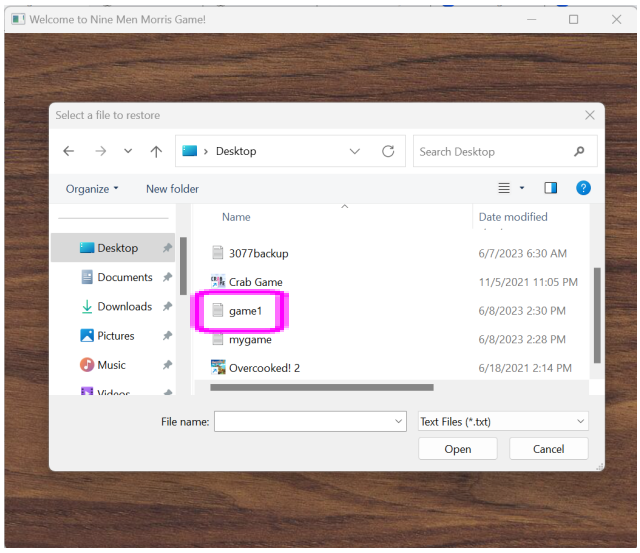
Players can save a game by clicking on the save button on the top right. They will be required to enter a name for the text file that will store the game states, then the text file will be stored in their desired location/directory.

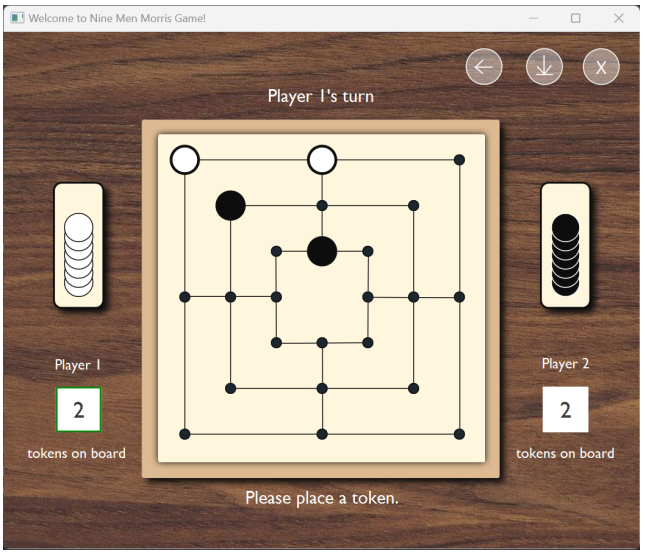




Players can also save the game before exiting. A message will pop up when players intend to quit the game, asking them whether they want to save the game or not.

b) Reload game

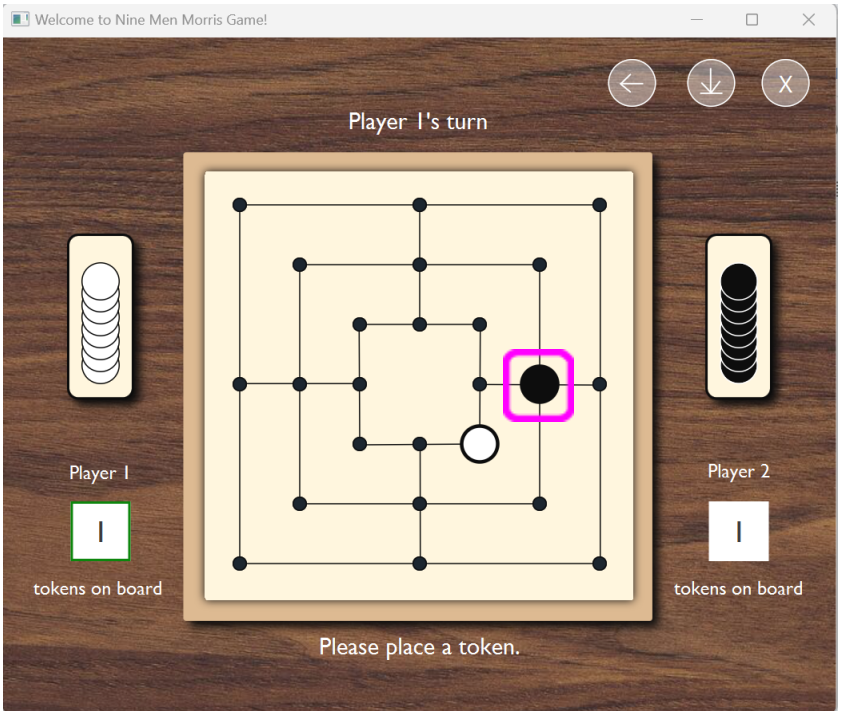




By clicking on the 'Load Game' button in the main page, the file explorer will appear so that players can choose the text file that stores the game information that they want to continue to play. By clicking on the text file, the game will restore the game states and players can continue to play the game.

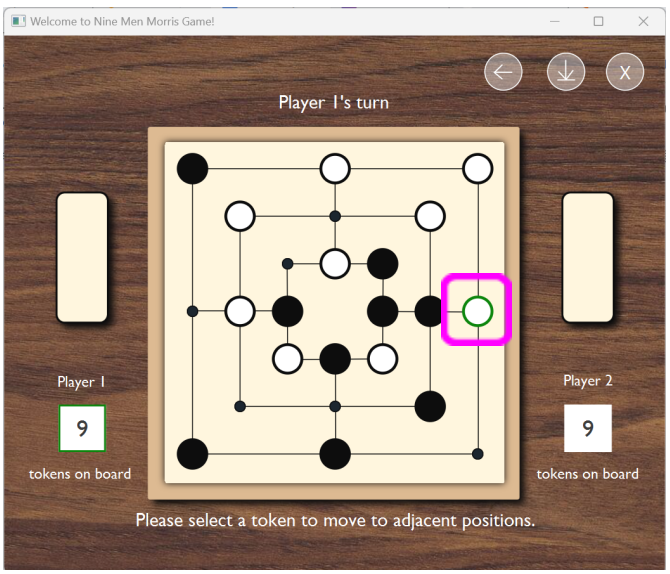
Playing with Computer

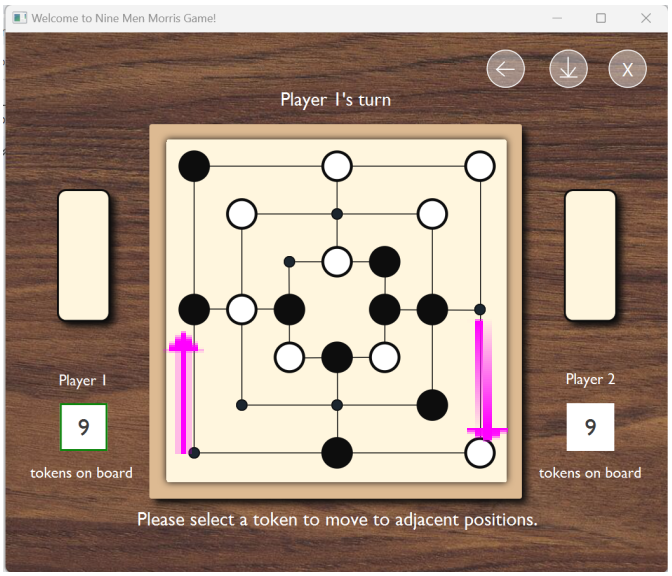
a) Placing token



Player 1 (white) places a token, then Player 2 also the Computer player (black) places its token in a random position that is empty.

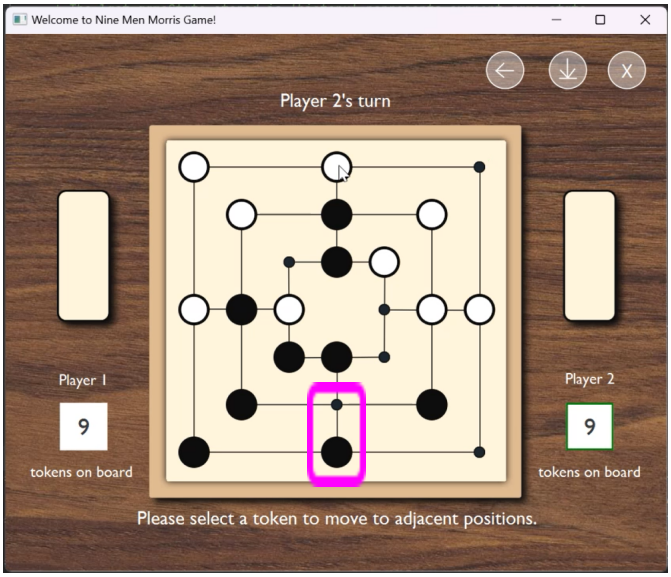
b) Moving token

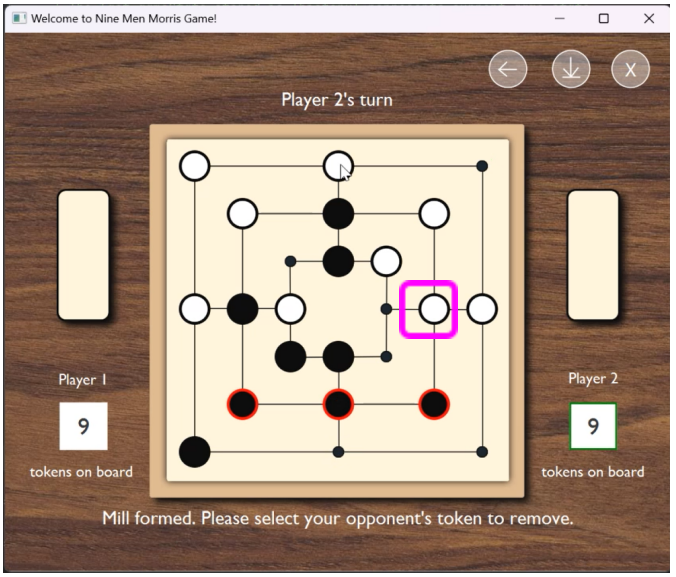


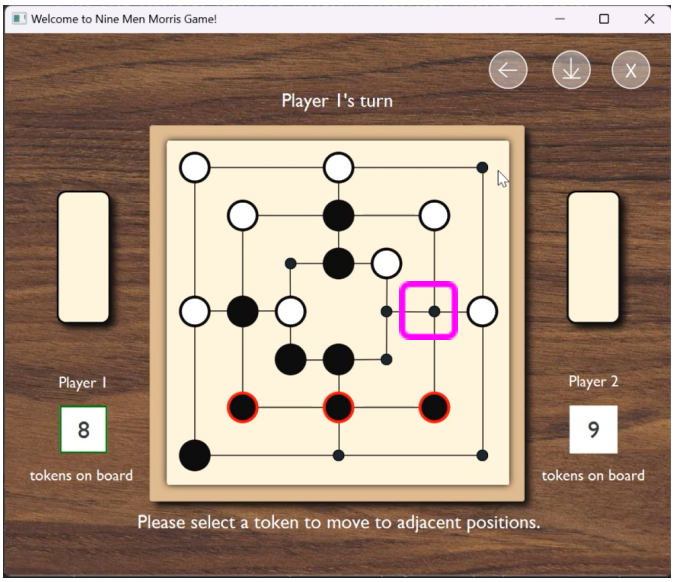


After Player 1 (white) moves its token, Player 2 also the Computer Player (black) randomly picks one of its tokens to move it to an adjacent position.

c) Removing token







When Player 2 also the Computer Player (black) forms a mill, it will randomly pick its opponent's token (white) to remove.