FIT3077 Sprint Two 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

Tech-based Work-in-Progress (software prototype)

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.

- Classes related to the advanced requirements have not been added to the source code yet (GameMemento, GameCaretaker, ComputerPlayer, etc). They are highlighted in blue in the class diagram.
- Classes related to BoardActions and ButtonActions have not been added to the source code yet as they are not related to the prototype requirements for Sprint 2.

As we are using JavaFx for this project, the game can be played in a pop up window that will appear when running the GameApplication class. Steps to play the game are provided below. As this is still a prototype, the game may run into unexpected behaviour if the user misclicks, such as double clicking on a token. That being said, the game has been tested and all tokens can be placed and moved to all positions if the user does not misclick anything, which fulfils the requirements of Sprint 2.

To set up the game, please refer to our README file on the main branch of our github.
https://git.infotech.monash.edu/fit3077-s1-2023/MA_Friday2pm_Team10/project

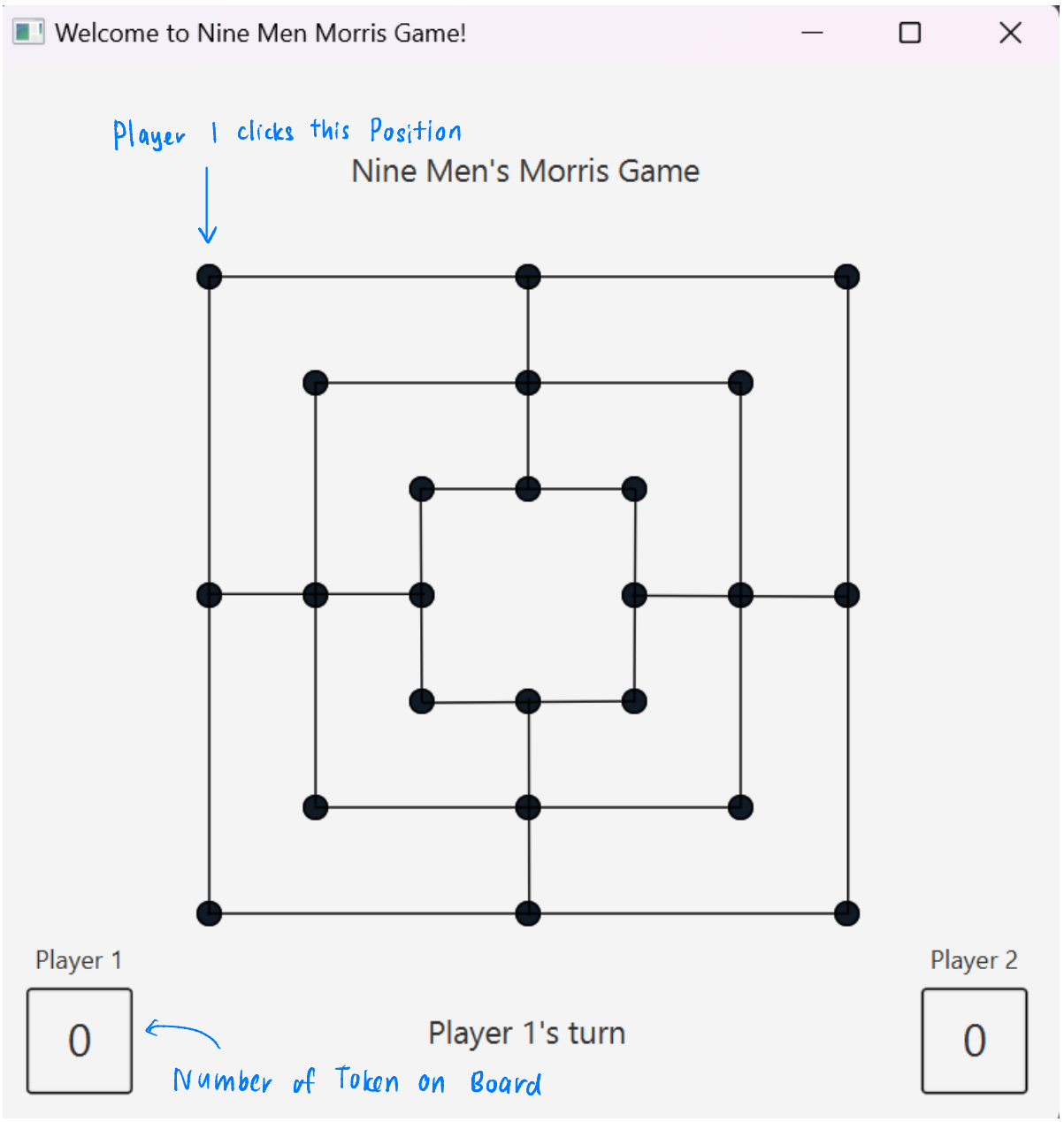


Fig 1. An empty board with 24 Positions (black circle) will be loaded. To place a token, Player 1 clicks on the Position (black circle) to place a Token.

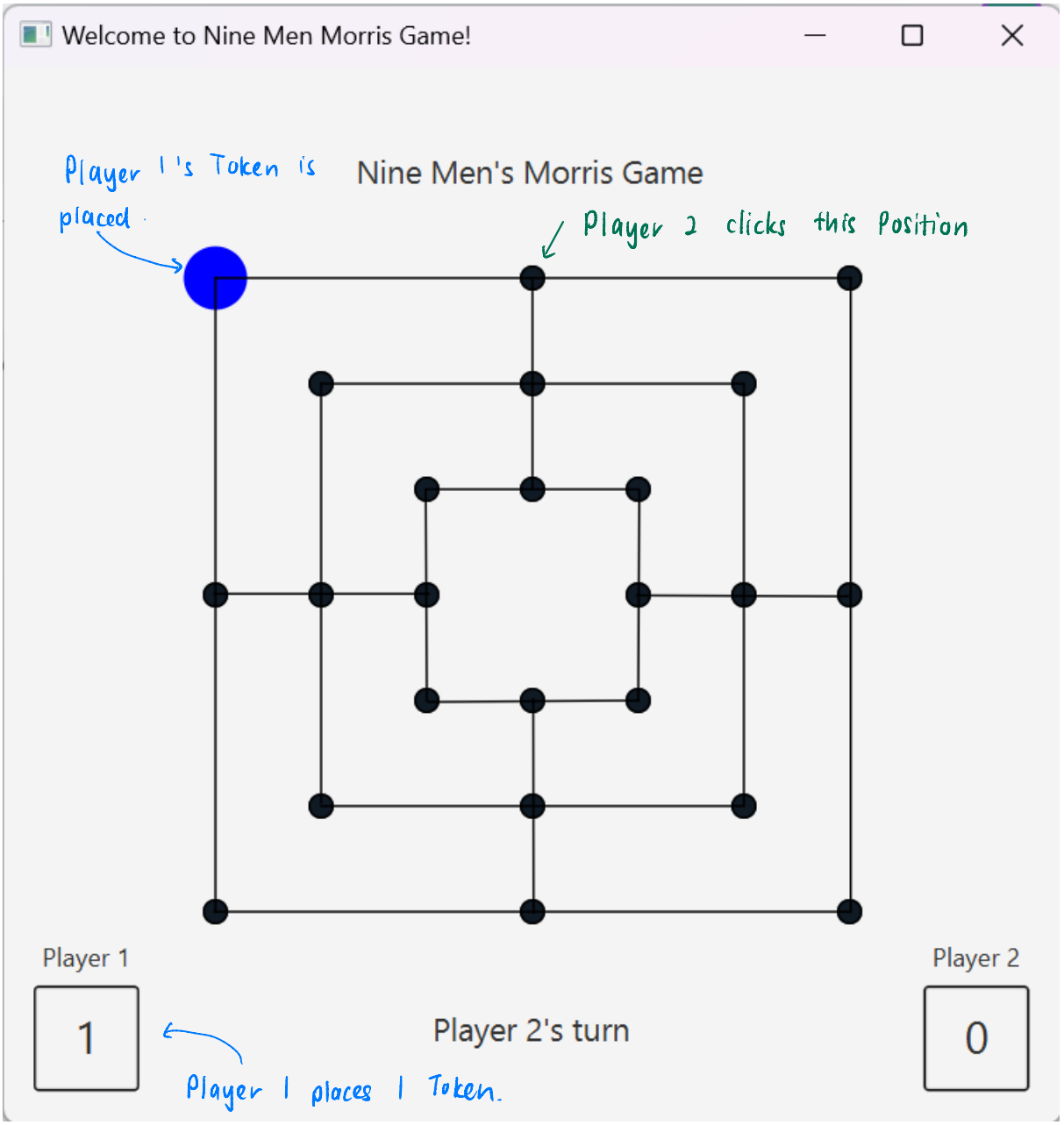


Fig 2. Player 1 places a Token on the top left Position. Player 1's Token is in blue. Player 1's number of tokens on board will be updated. After Player 1 places a Token, it will be Player 2's turn.

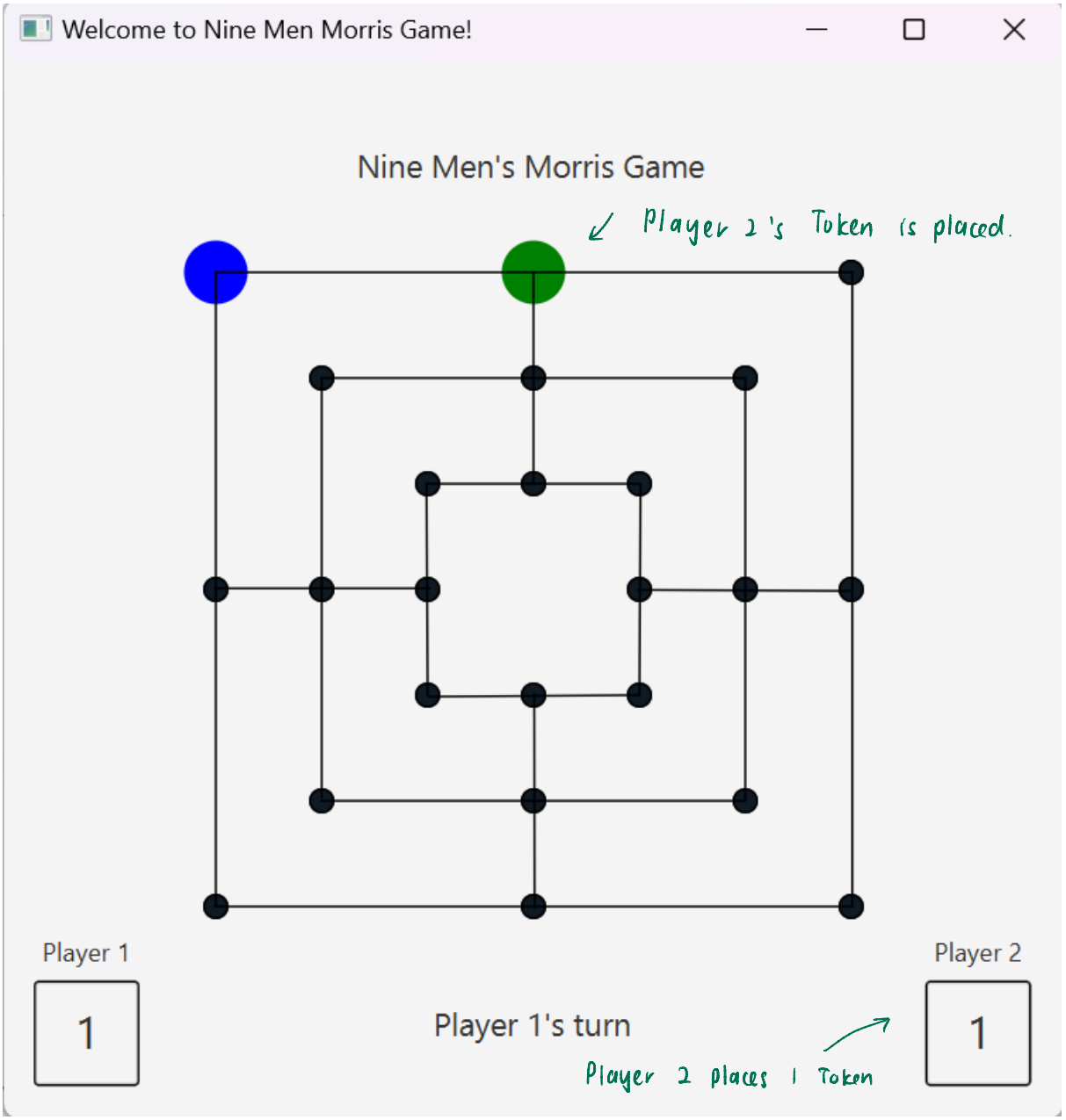


Fig 3. Player 2 clicks on the second empty position and places a token. Player 2's Token is in green. Now, it is back to player 1's turn to place a token.

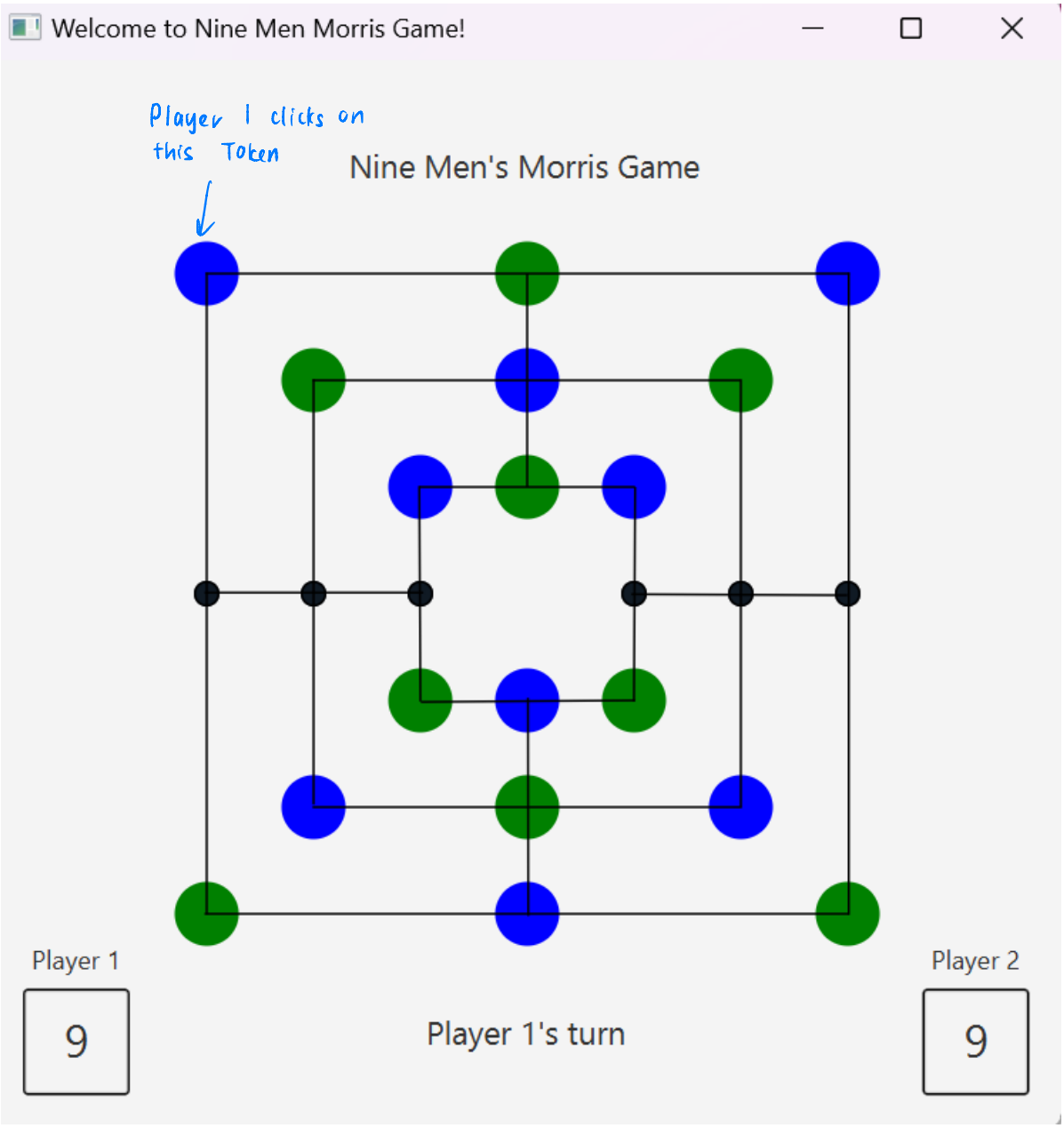


Fig 4. Both players have placed their nine tokens on the board. They can start moving their tokens.

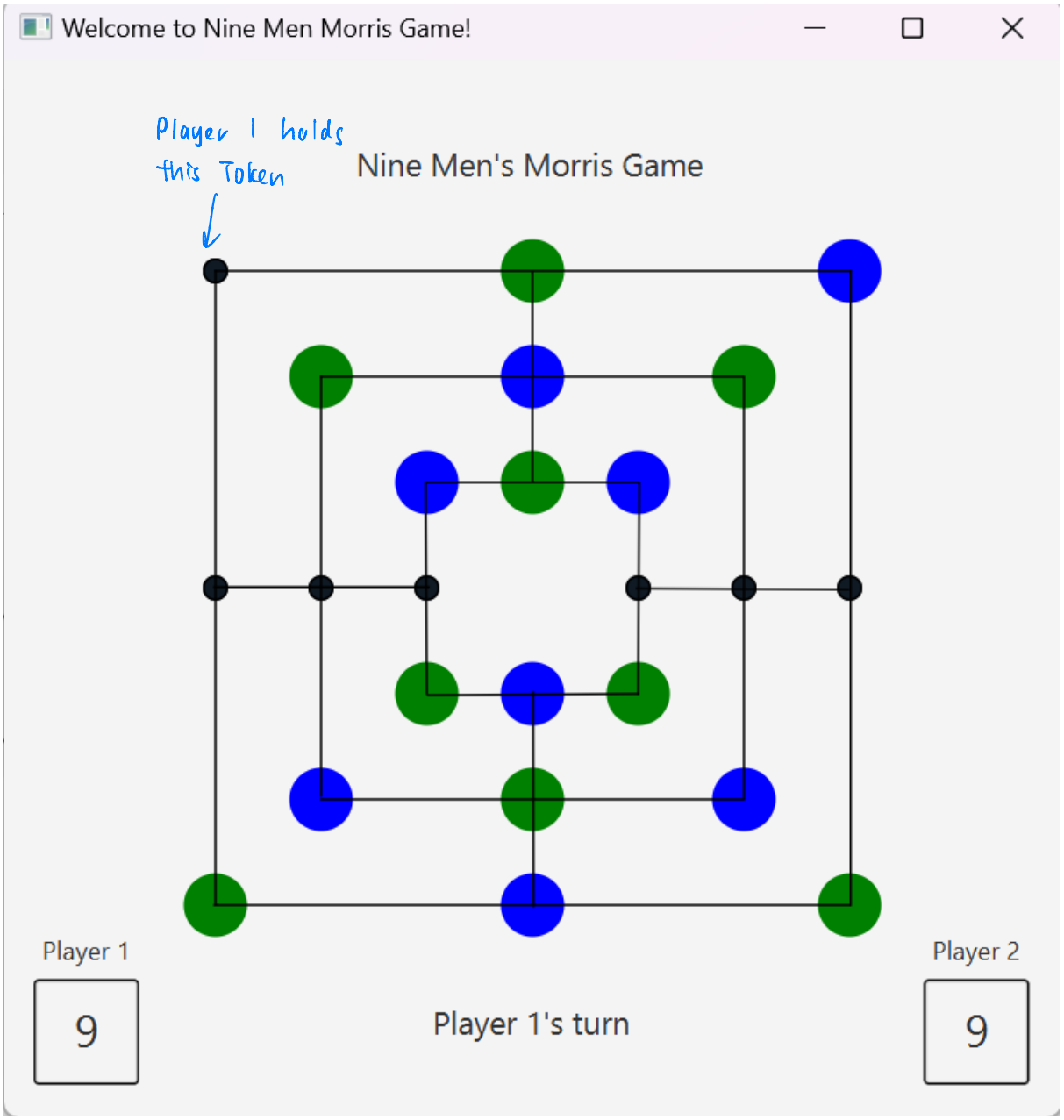


Fig 5. Player 1 clicks on the top left corner token (blue circle), which makes it hold onto it, hence it is removed from the board.

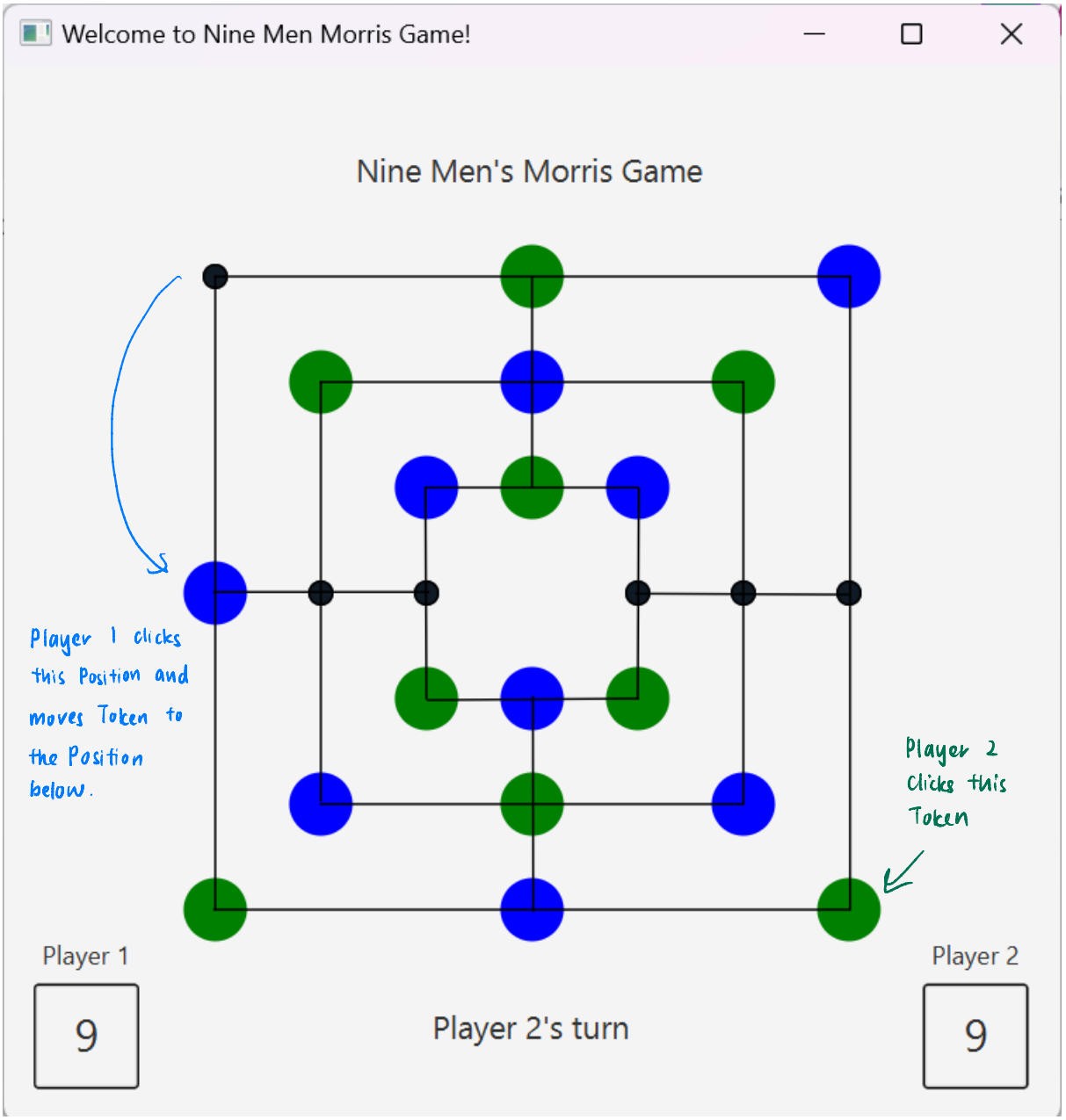


Fig 6. Player 1 moves the Token to the Position below it by clicking on that Position. The token on hold is now placed on the new position. Now it is Player 2's turn to move a token.

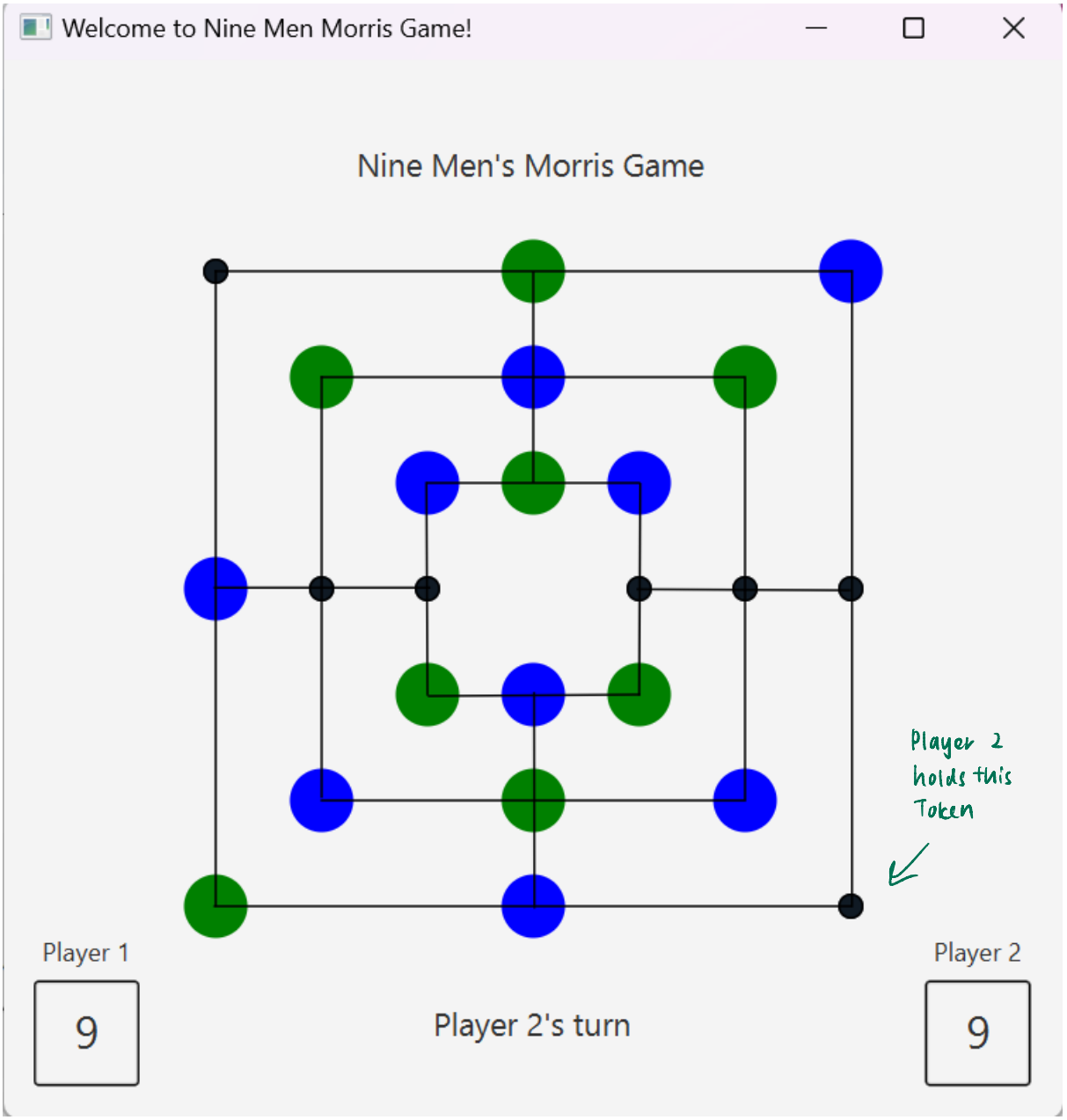


Fig 7. Player 2 clicks on the bottom right corner token, which makes it hold onto it, hence it is removed from the board.

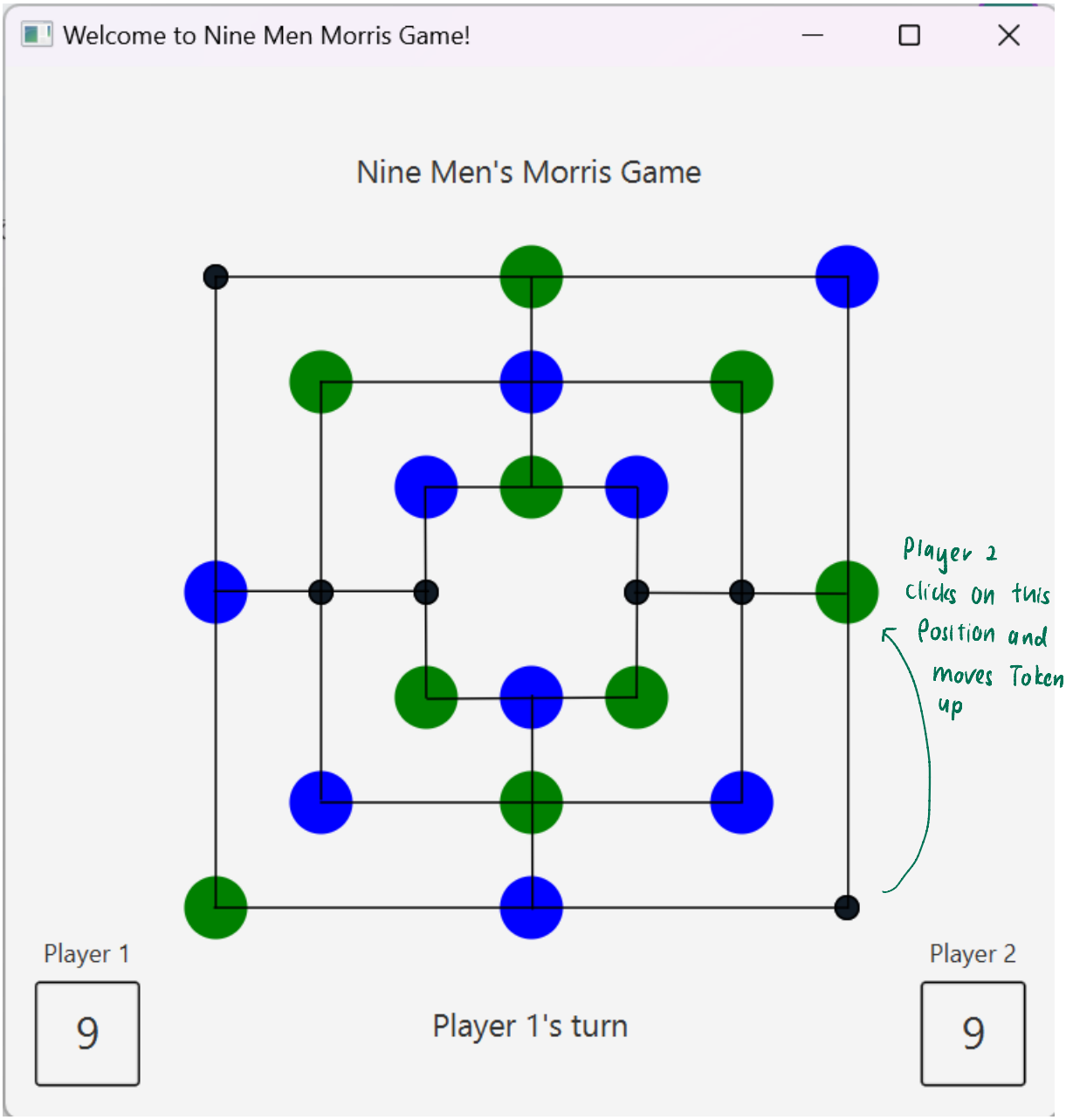
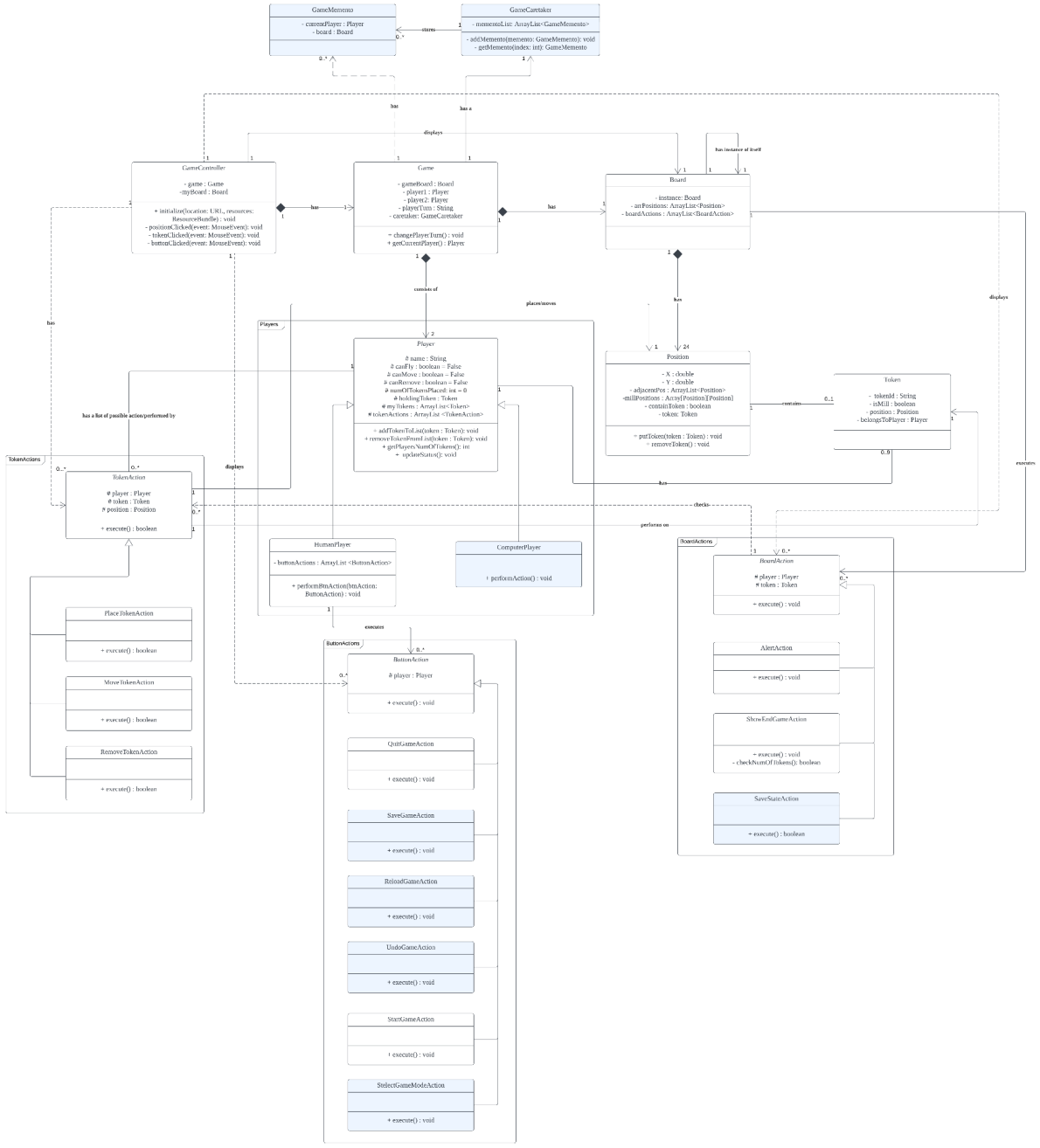


Fig 8. Player 2 moves the Token to the position above it by clicking on that empty position. Now it is back to Player 1's turn to move a token.

Architecture and Design Rationales

Class Diagram:

https://lucid.app/lucidchart/e66943cd-742a-4ce3-8dd3-ba4812ec3b4b/edit?viewport_loc=195%2C-627%2C2574%2C1235%2CAGAOT.Fznp-P&invitationId=inv_52e98539-aab1-4b23-8d36-c80cdd648ae7



Design Rationales:

Key Classes:

1. Position and Token.

   Each position can store up to 1 token. This logic can be implemented by simply using a boolean variable containsToken which would be true if there was a token at the position and false otherwise. Thus, we simply used a setContainToken method to achieve the functionality mentioned above. However, we decided that Token should be a class on its own because Token is a separate entity with its own attributes and behaviour, such as knowing which player it belongs to and its respective colour. While we could have tokenColor and tokenPlayer attributes and implement setTokenColor and setTokenPlayer methods inside the Position class, that implementation would be in violation of the **Single Responsibility Principle**. In conclusion, we created a Token class to encapsulate the data and behaviour of tokens rather than implementing methods inside the Position class.

2. Player and TokenAction Classes(PlaceTokenAction, MoveTokenAction, RemoveTokenAction)

   A Player can make different types of moves such as placing, moving, flying, and removing a token. As these actions are considered as behaviours that belong to the player, one option is to encapsulate these behaviours as methods inside the Player class. However, these actions are complex as they involve Position, Token and front end components(GameController class). It makes sense for us to extract these complicated methods that share similar data into a separate class/classes. Implementing these actions inside the Player class directly would violate **Single Responsibility Principle** and also cause Player class to become a "God Class". Thus, our solution is for each action to have a class on its own to keep the logic separate and the code modular, which allows for better code maintainability.

Key Relationship:

1. Position and Token class have a bidirectional association between them. This means that each Token knows its current Position, and each Position knows which Token it contains. This bidirectional association is necessary for several reasons.

   Firstly, when a player wants to move a Token from one Position to another, the game needs to know the current Position of the Token to check if its adjacent positions are empty to determine whether the move is valid. Thus, Token has an association relationship with Position.

On the other hand, the game needs to be able to save the game's state so that players can undo moves or reload previous games and resume playing. Therefore, Position needs to be associated with the Token in order to know if a Token is present or not. Thus, Position has an association relationship with Token.

2. Player and Token class have a bidirectional association between them. Each Token knows which Player it belongs to, and each Player stores an array list of their tokens. The game rule requires this bidirectional association.

   The game needs to keep track of which Token is owned by which Player. For example, we need to be able to identify who the token belongs to when moving tokens as a player should not be able to move an opponent's token. Similarly, when a player forms a mill, we need to allow the player to remove the opponent's token. We need to be able to identify that the token that is going to be removed belongs to the opponent. For the reasons above, Token is associated with Player.

   On the other hand, Player has an array list of their Tokens. The game loops through the tokens that a player has, and checks for valid moves. This is needed to check for the winning condition, where a player can no longer make any moves. Thus, Player is associated with Token.

Explanation about Inheritance:

Inheritance represents an "is a" relationship between the children and parent class. For example, we have used inheritance where PlaceTokenAction, MoveTokenAction and RemoveTokenAction inherit from the abstract parent class, TokenAction. As these 3 Action classes have similarities such as requiring Player, Token and Position classes, using inheritance allows us to reduce code duplication, which is in line with the **DRY Principle**. Another advantage of using inheritance is polymorphism. Due to the abstract execute method inside the TokenAction class, each child class can override the execute method to perform its respective role. By using the abstraction provided via inheritance from the TokenAction class, we also abide by the **Dependency Inversion Principle**. (We can store an arraylist of TokenActions instead of using 3 separate array lists for each type of action).

Cardinality Explanation:

1. Position and Token class. A position may be empty, so it contains 0 tokens. If a token is placed on an empty position, then the position contains 1 token. Tokens cannot be stacked, so a position may contain up to 1 token only. Thus, the cardinality of Position and Token classes is 0 to 1.

2. Player and TokenAction class. A player stores a list of token actions. For each token the player has, we check the valid moves that can be performed on that token. All the token actions that can be performed are stored inside the list mentioned. This is needed because we need to end the game when a player cannot make any moves. The list may be empty if a player cannot make any moves, and the list may contain a lot of possible moves too. Thus, the cardinality of Player and TokenAction classes is 0 to many.

Design Patterns:

For Sprint 2, most of the game logic has not been implemented yet. Hence, after considering the trade-off of implementing design patterns, we selected some of the design patterns that we think will improve the game architecture and at the same time will not increase the complexity of our code. Implementation of design patterns are still open for changes during the upcoming sprints to improve our game design and architecture.

1. Singleton Pattern
   A singleton is a design pattern that ensures that only one instance of a class is created and this instance can be accessed globally. Since a game only has a game board, we will implement the game board as a singleton instance so that both players can play on the same board. This design approach ensures that there is only one board instance throughout the game and allows us to access the game board easily from anywhere without passing it as an argument.

2. Two alternatives discarded:
    1. Factory Method Pattern
       It is used to encapsulate the creation of objects and to provide a way for subclasses to choose which class to instantiate. It was initially considered to be implemented by creating a PlayerFactory interface, and having HumanPlayerFactory and ComputerPlayerFactory that implements it to create the corresponding Player objects. However, considering that additional types of players will not be introduced into our game, we think that implementing this design pattern will introduce complexity to our code instead of improving our design architecture.

    2. State Pattern
       It is used to manage the different states in our game such as Place Token State where each player places 9 tokens onto the game board, and Move Token State where players move their tokens to form mills. We initially considered implementing a GameState interface, with PlaceTokenState and MoveTokenState class implementing it. Implementing the state pattern will improve our code's maintainability as when new states are added in the future,

we can add another new class that implements GameState interface by only having to make minimal changes to existing code. However, our game will only have two states which are Place Token State and Move Token State, hence implementing the State pattern will not bring benefit in the maintainability aspect but will introduce complexity to our code.

Acknowledgement

We acknowledge the use of ChatGPT (https://chat.openai.com/) for some design pattern suggestions in the drafting of this assignment. We entered the following prompts on 24th April 2023:

- Most commonly used OOP design patterns for nine man morris game construction
- What are the advantages of inheritance?
- Any other structural patterns?
- Any other Behavioural patterns?
- If my game has an undo function, how should I implement memento?
- How to use the singleton principle in 9mm?

The output from the generative artificial intelligence was discussed among the team and modified for the final response.