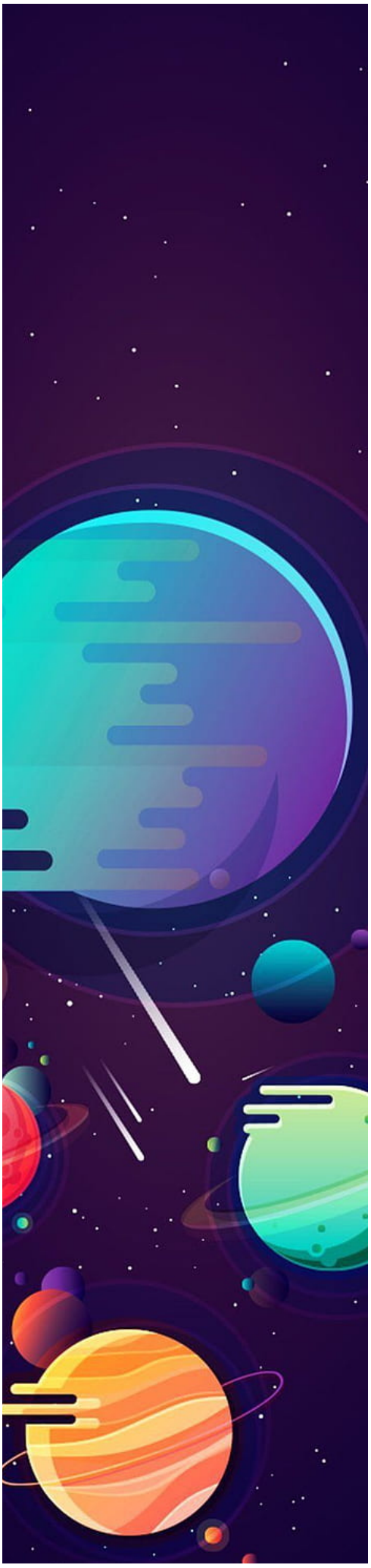
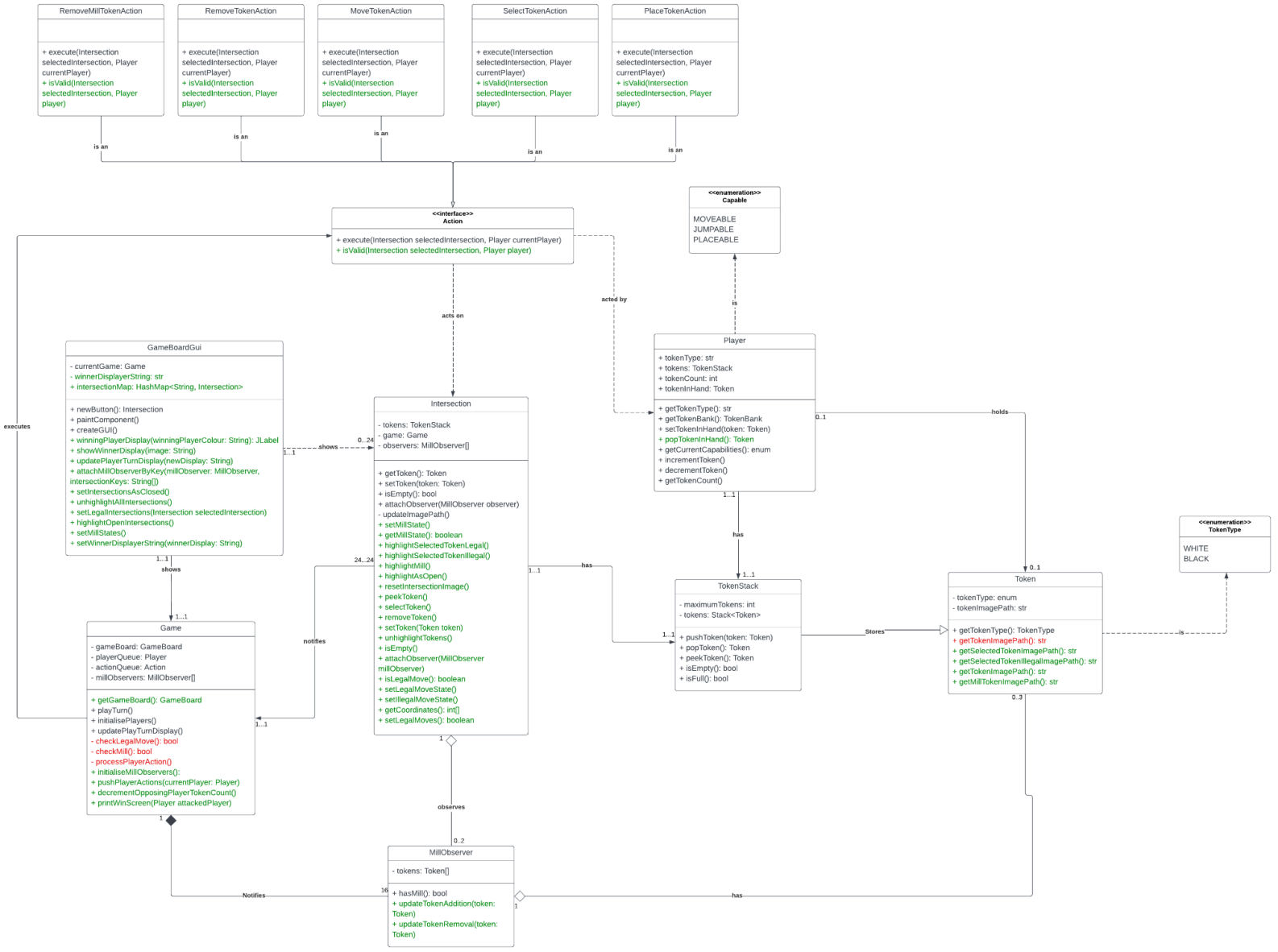


FIT3077
Sprint 3
Design Rationale

Table of Contents

Revised Class Diagram



Design Rationale

Revised Architecture

Action Interface: isValid()

Issue:
When an action is being made, the game must check if the action being executed can be done.

Alternative A:
Have the game class check every move that is being made when the player is playing a turn

Advantages:
- One method that effectively checks all the moves being made when the player executes a turn
- Simple implementation and easy to code

Disadvantages:
- Will be a massive method with a lot of unnecessary code when one particular action is being checked
- Harder to test as the method accounts for all the action classes that have to check for different validities

Alternative B:
Implement a method into the action class that allows each action to verify its own validity before executing

Advantages:
- Easier to test because each action class is responsible for its own valid checking
- Smaller code for each action as it only needs to check for the particular move its making e.g the move class must check if the token being moved is to an empty intersection

Disadvantages:
- A bit more work to implement since you have to code essentially the same code with little changes in each class
- Have to call the isValid() method for each class

Decision:
Alternative B was chosen for this implementation as the team decided it will be easier to test the code for errors and bugs if the code that handles the action for each type of action was separate in its own class. Hence the isValid() action was added to the Action Interface in which the other classes inherited. Though each class must run the isValid() function, it is done in its own respective class which leaves the Game class to handle its own functions without taking on too much responsibility thus avoiding becoming a god principle. This decision ultimately follows Liskov's Substitution Principle as wherever the Action interface is called, the game can call the valid method using just one statement of code.

Concrete Action: RemoveTokenAction and RemoveMillTokenAction

Issue:
Typically, when removing a token from an opponent, tokens in mills are protected. However, if all of an opponent's tokens are in mills, they are not protected. Where should this logic be handled?

Alternative A:
Have a single RemoveTokenAction concretion of the Action interface, which handles removing both milled and unmilled Tokens, and verify which input is appropriate in the Game's playTurn() method.

Advantages:

- Maintains just four Actions, place, select, move and remove, which reflect the four intuitive interactions that a user can have with a physical game of 9MM.
- Game has access to the GameBoard, which, in the current design, has the cleanest means of checking whether all of a user's tokens are milled by querying its intersectionMap hashmap. Since this logic is already occurring at Game, it is not unreasonable to add more logic at the same scope.

Disadvantages:

- Leads to further bloat for Game's playTurn() method, which, as the primary game engine loop method, is already large, posing readability and testability concerns.
- Necessarily means that the isValid() method in the RemoveTokenAction will need to be combined with the Game's logic to detect the number of mills, separating the logic for validity into multiple objects of the codebase.

Alternative B:
Add a RemoveMillTokenAction that extends the RemoveTokenAction, using the same execute() method but an overridden isValid() method that does not reject selecting Intersections with mills.

Advantages:

- Relatively lightweight, as the overridden isValid() method is simply a very short conditional that is clearly readable.
- Minimises the need for additional logic in the Game class while following the same convention as other Actions.
- Keeps more of the validation logic in the same part of the code, improving readability and providing a clear precedent for the extension.

Disadvantages:

- Game would have to handle determining which Action to distribute: A RemoveTokenAction or a RemoveMillTokenAction. This logic would add a comparable amount of bloat to Game to that which would be added by having Game handle validity.
- In general, the design choice made by the team has been to favour composition over inheritance. However, this is a case where inheritance is a clear winner, allowing for clean code but potentially muddying the design direction and making extensibility options less clear.

Decision:
Alternative B was chosen in order to minimise the amount of bloat added to the Game class, even if the difference is marginal, and to reinforce the paradigm of using Action's isValid() method to determine move validity. Despite this, the logic for move validity remains somewhat shared across Actions, which determines which Intersections states are legal to select (in this scenario, depending on whether the Intersection is part of a mill), and the GameBoard, which queries Intersections that are accessible from a given Intersection in order to determine whether any of the accessible Intersections meet the isValid() criteria. This division of logic is an area for discussion in Sprint 4; however, the separation of validity logic into two existing classes does not justify further splitting it into three by having Game handle additional validity checks.

The addition of an Action specifically for removing Tokens from milled targets is a clean solution that extends easily from RemoveTokenAction and is clear enough conceptually that it is not expected to pose any significant increase in complexity and reduction in readability, and keeps validity checking in one of two places - Action or GameBoard.

GameBoardGUI: Winner Display Function

Issue:
Showing which user has won the game

Alternative A:
Have a Jbutton that remains invisible over the top of the gameboard and only activates when the game is lost by a player

Advantages:
- JButtons are very simple to manipulate hence having an invisible button that gets fed an image path when a player loses is very simple
- Only need to access one variable of the GUI to replace its image path

Disadvantages:
- Button will be clickable once it appears which will confuse the user and decrease usability as this shouldn't be clickable

Alternative B:
Make the JLabel appear when a player has lost the game hence only being added to the GUI when a player has lost

Advantages:
- Meets the usability as the JLabel only appears when the game has finished and ceases the user from clicking anything else
- Meets the Open Closed principle as the label's image path is private and can only be set by the setter

Disadvantages:
- Requires a few classes to implement and a bit complex
- Slow to appear

Decision:
Alternative B was chosen for this implementation despite its complexity as this adheres to the reliability aspect of the non functional requirements by making the image path only be manipulated by a setter. This in turn fits the open-closed principle as when the label appears, it should only signify the end of the game making it difficult for the label to be edited. This also makes it simple for the Game class to pass the string of the player who has won and the display method will make that the image path.

GameBoardGUI: Coordinate system and Hashmap vs graph implementation

Issue:
Should the connections between intersections be represented through a coordinate system and hashmap or through a graph?

Alternative A:
Have the intersections be represented as a graph with four connections for each edge.

Advantages:
- Additional functionality can be added through the graph implementation

Disadvantages:
- Decreased performance as might have to traverse from source node to last node, resulting in complexities up to O(E) time complexity where E is number of edges
- Implementation would be quite difficult in order to accurately reach required node

Alternative B:
Have the intersections be represented in a hashmap and coordinate system.

Advantages:
- Substantially better performance, as hashmap takes O(1) time complexity, and whole intersections can be accessed with key.
- Relatively easier to implement once coordinate system is established

Disadvantages:
- Harder to maintain as search algorithm is quite complex and would not be easy to understand from first glance

Decision:
Alternative B was chosen primarily for performance reasons and simple implementation reasons. Although alternative A has better extensibility through a graph implementation, overall the performance advantages and easy to implement pros for Alternative B made it the most viable choice. Intersections are accessed every turn thus good performance is integral, thus making Alternative B the clear choice.

Player: Should player hold its own tokencount or should the game keep track

Issue:
Should player have their own tokencount variable or should the game keep track of both players token count

Alternative A:
Player keep track of own token count

Advantages:
- Allows for extensibility of code as more players can be added without having to refactor game to keep track of players
- Simple implementation and easy to code

Disadvantages:
- Additional steps required to retrieve other player and invoke their tokencount decrement

Alternative B:
Game keep track of both players token count

Advantages:
- Easier to implement as game can see both players token counts at once, and can decrement tokencount easily

Disadvantages:
- Provides no extensibility, as each time a new player is added, a new tokencount for that player will have to be added
- Can only access token count from Game, meaning other classes that have access to player cannot get the players tokencount without directly requesting from game.

Decision:
Although alternative B was a lot easier to implement, Alternative A was chosen. Alternative A adhered to OOP principles more closely, allowing for extensibility, as if more players were to be added, the game would have to create a new token count every time, which makes it extremely inextensible. Whilst if the player kept track of their own tokens, then new players can be added with no additional refactoring. Thus although Alternative B is a lot easier to implement, the advantages of Alternative A made it the most viable choice.

Non-Functional Requirements

Usability

Quality Attribute:
Usability

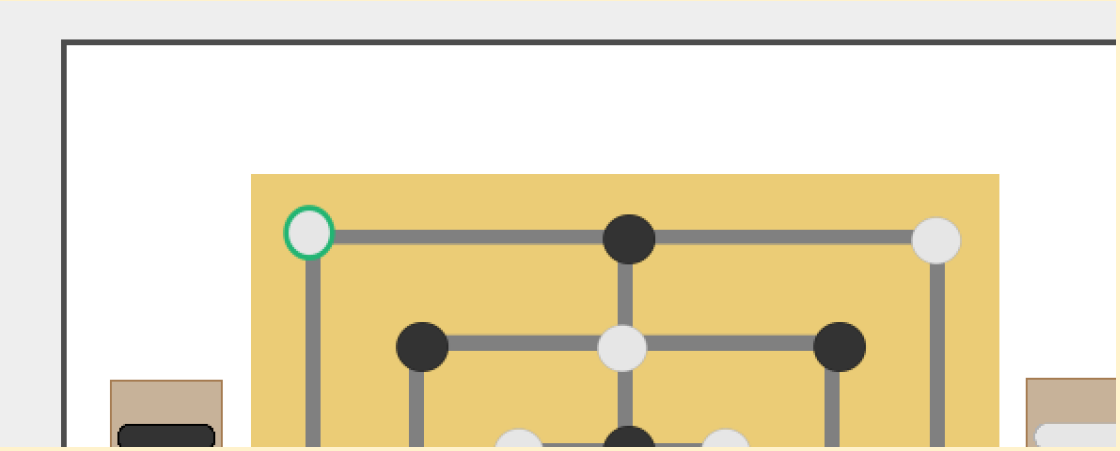
Considered In Design:
Nine Mens Morris game should incorporate usability most important to their users as a non-functional requirement. Since the game is predominantly a front-end user interface that the player must interact with, it is important to make sure the game is easy to play and understand just by using common cognitive thinking.

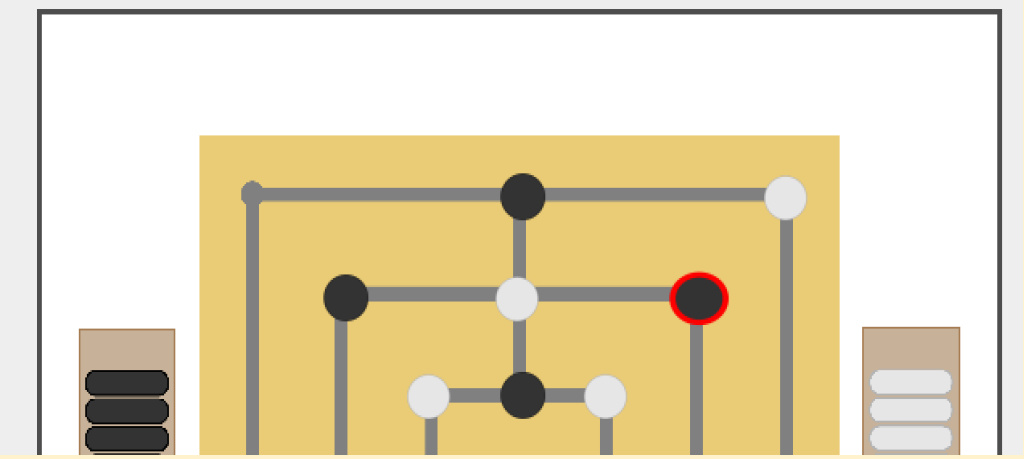
In our implementation, the game shows when a token is selected in colours that adhere to cognitive thinking. If a token that can be moved is clicked it will have a green ring around it which indicate it can move, likewise, it will have a red around it to show it cannot be moved and the player may select a different token.

Importance:
Usability in the tokens changing colour brings about user satisfaction by making the game easier to play and to understand if a player is new to playing Nine Men's Morris or simply trying to understand. The user satisfaction criteria are also promoted by this feature as the user will find it easier to navigate the game and get feedback on the selection they make without much difficulty.

The addition of the coloured token selection aspect also adheres to accessibility and inclusivity as it caters to all audiences in different age groups and different accessibility needs in understanding the game and playing without having played before or knowing the rules. This makes the game usable for all demographics and overall provides a better user experience.

Evidence:





Reliability

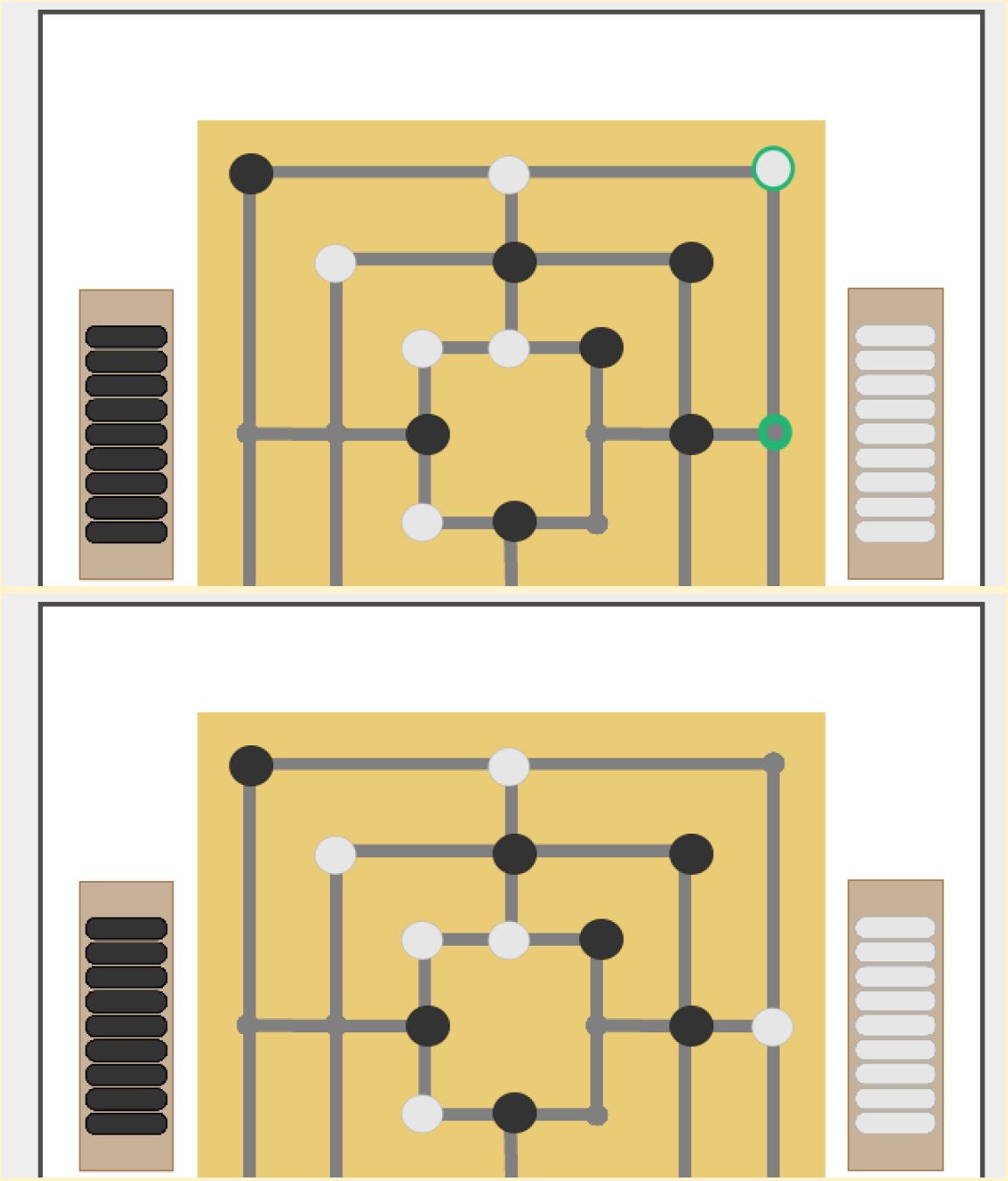
Quality Attribute:
Reliability

Considered In Design:
Reliability was considered in the design as a key non-functional requirement as the game solely depends on being played properly if the rules are implemented the way the player expects them to be. Consistent gameplay is something our implementation has heavily followed by closely applying the nine men's Morris rules as per several found references.

One of the most important rules the game has implemented is each player only being able to move a token one space vertically or horizontally from their current location. This allows the game to be played fairly and accurately.

Importance:
The importance of this rule is that if the tokens can be moved to any place the player wants to, it leaves for inaccurate and mundane gameplay. As the game dons the name “Nine Men’s Morris”, it is important to align accurately to the rules and players will not want to play the game if it is not to their expectations. This also hinders user satisfaction as the user will not trust the gameplay to be fair. Reliability also ensures that these interactions are consistent each time the game is played, preventing unexpected errors or inconsistencies in the gameplay.

Evidence:



Human Values

Achievement

Human Value:

Achievement

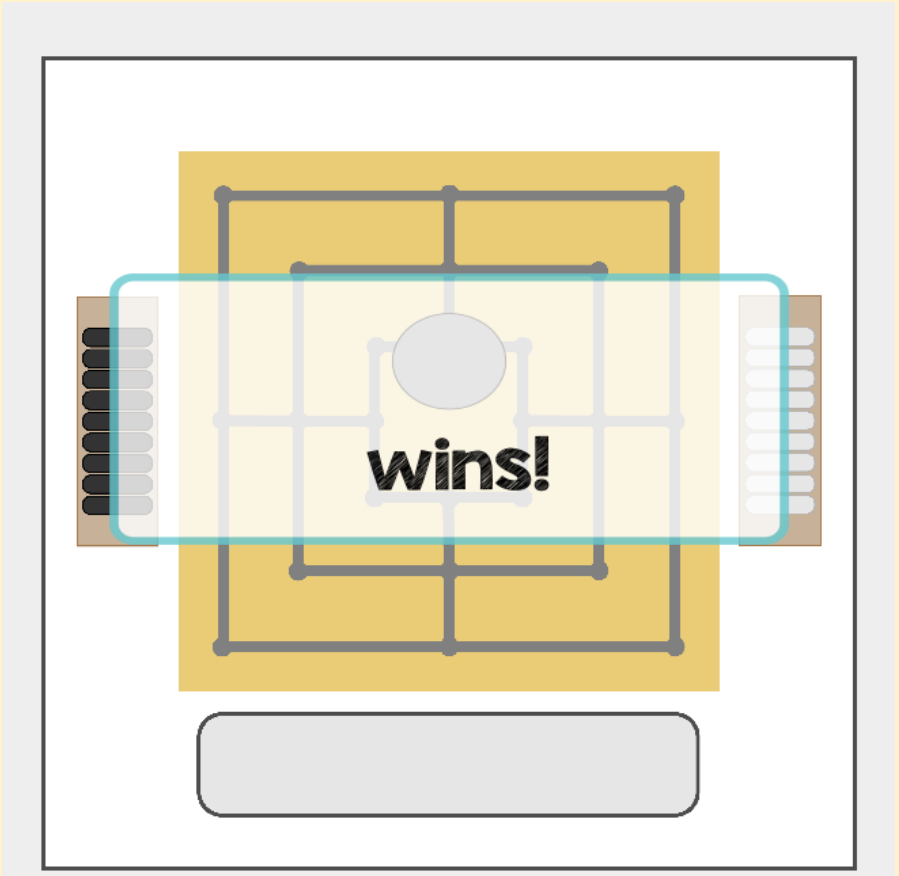
Considered In Design:
Achievement was considered in the design for game completion. While the simplest solution would have been to simply stop allocating Actions upon a Player losing the game and allowing users to interpret the winner, a decision was made to overlay a large visual declaring the winner in order to reward the winning user and appeal to their drive for achievement by celebrating this victory with them. A pleasant font is used with an exclamation to emphasise the value of the user's achievement. In doing so, it is hoped that users will be more likely to feel fulfilled by their gameplay.

The use of pleasing visuals also appeals to the human value of hedonism, which sits adjacent to achievement on the Schwartz Theory of Human Values' value circle. This implies that users who play Nine Mens' Morris for a feeling of achievement may enjoy the pleasure of aesthetically pleasing gameplay, which could influence Sprint 4 additional feature design such as adding auditory cues for victory and mill formation as a form of hedonistic reward for the user. However, the Sprint 3 design focuses on achievement over hedonism and so will not be discussed further.

Importance:
As a competitive game, Nine Mens' Morris is an instrument for human players to challenge themselves and others. Success in a competitive environment is akin to a demonstration of competence to oneself and others in the context of meeting a social standard; success at a video game allows users to show themselves and others that they are capable of reasoning and planning in a strategic game at a level that is similar to that of their opponents, helping to improve their self-esteem and social status.

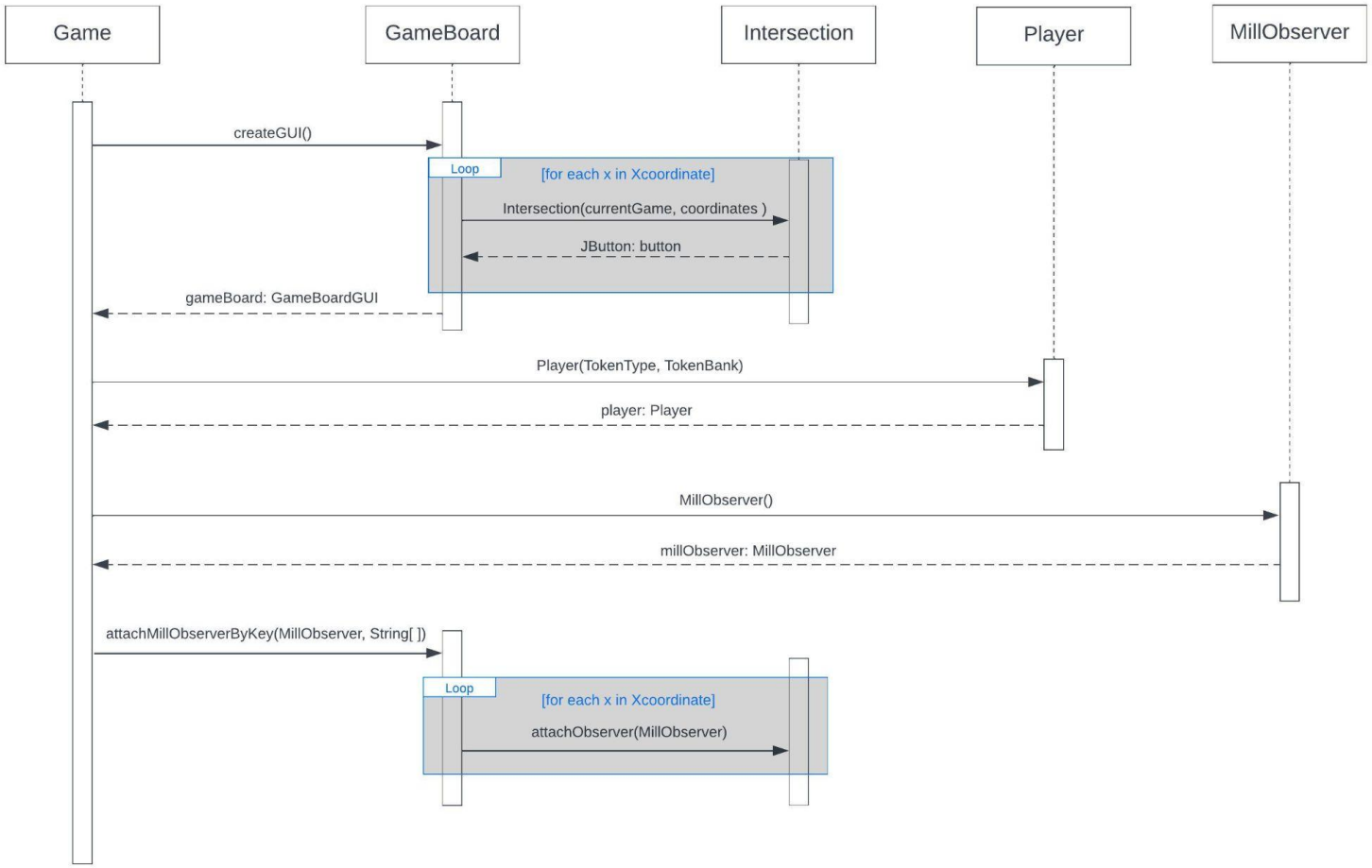
As such, it is critical that the Nine Mens' Morris design take achievement into account and ensure that the user's achievements are rewarded and clearly celebrated. In doing so, users across cultures with a drive to compete will receive the gratification of victory they expect and are more likely to enjoy and return to the game.

Evidence:

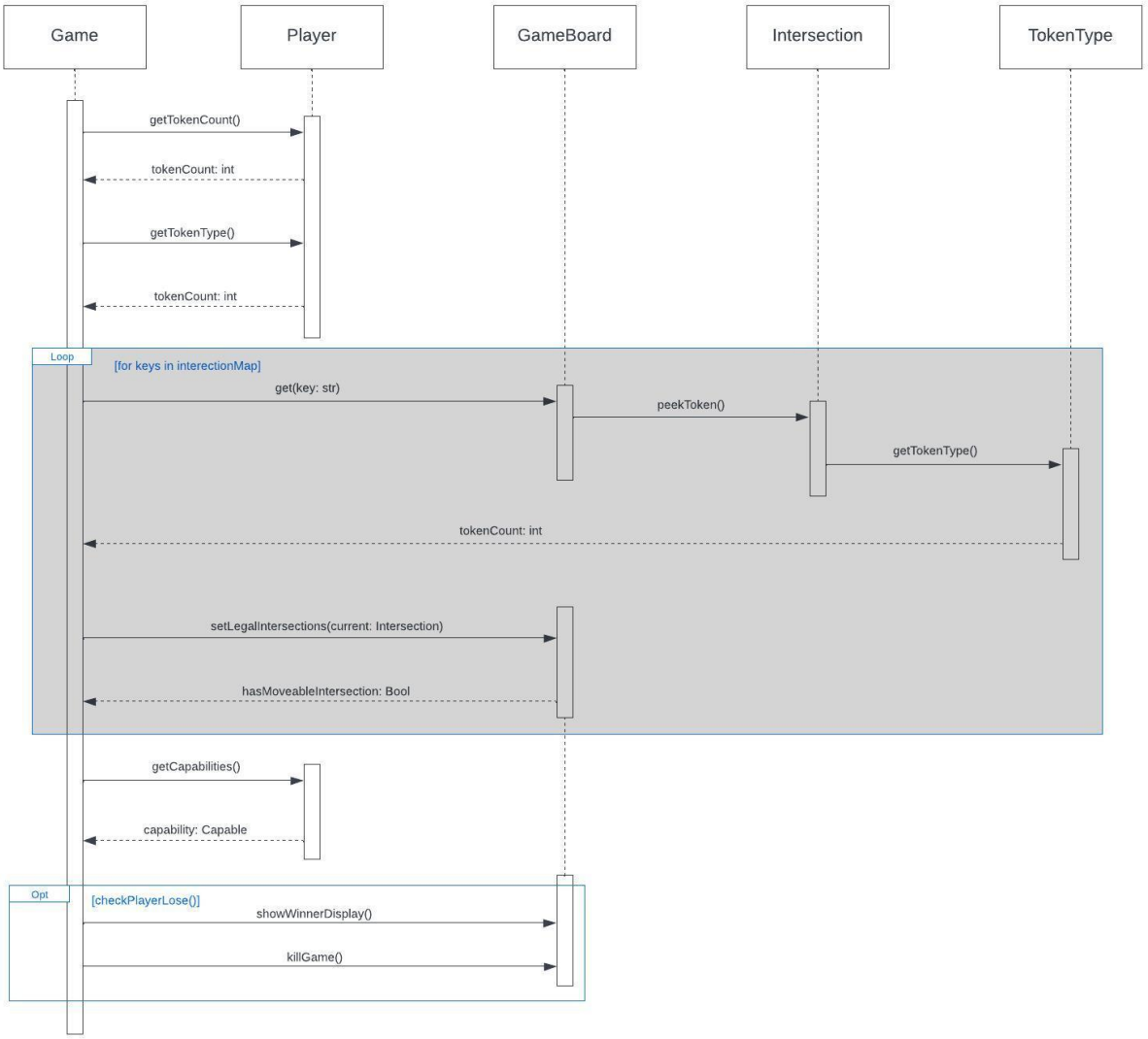

wins!

Sequence Diagrams

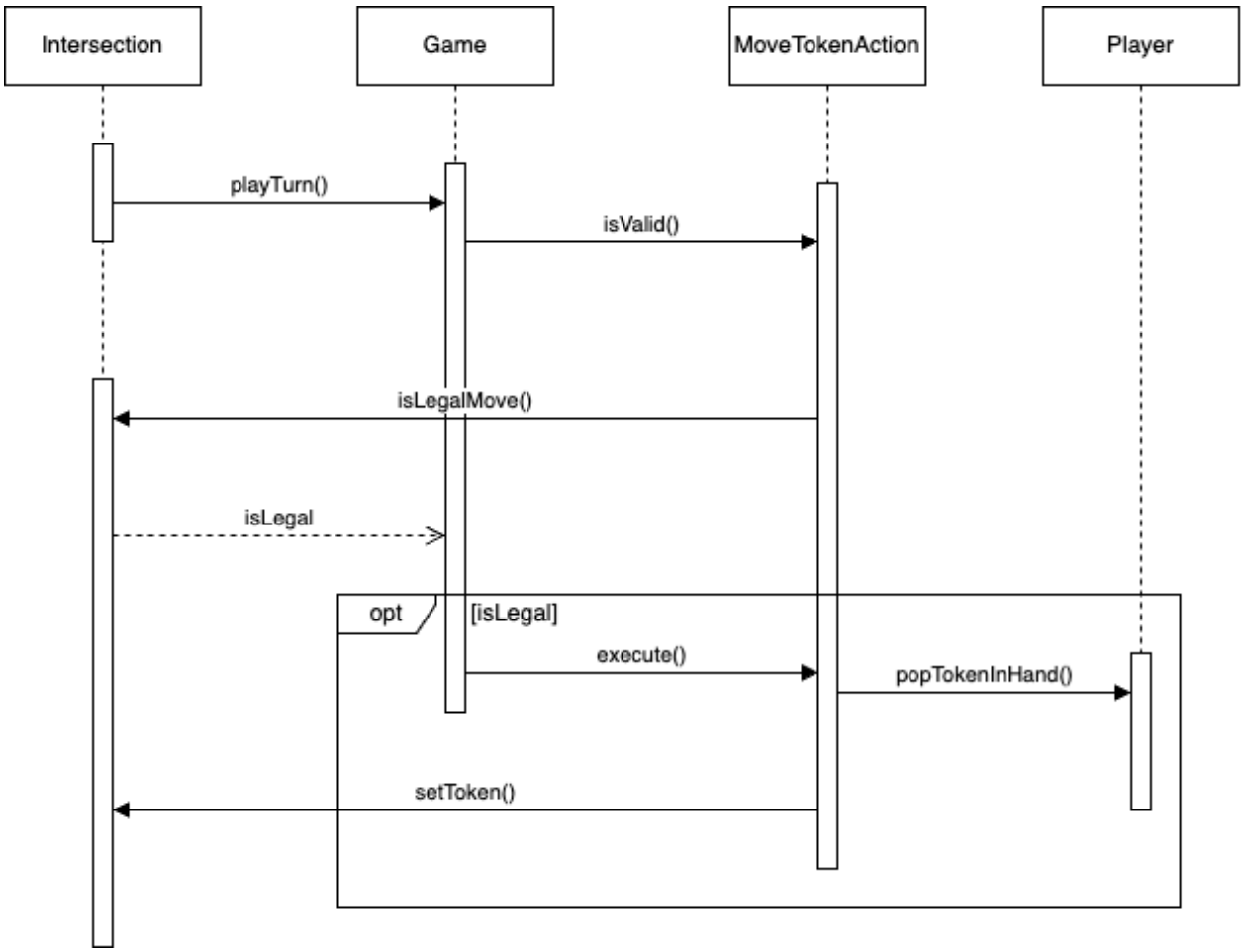
Game Initialisation



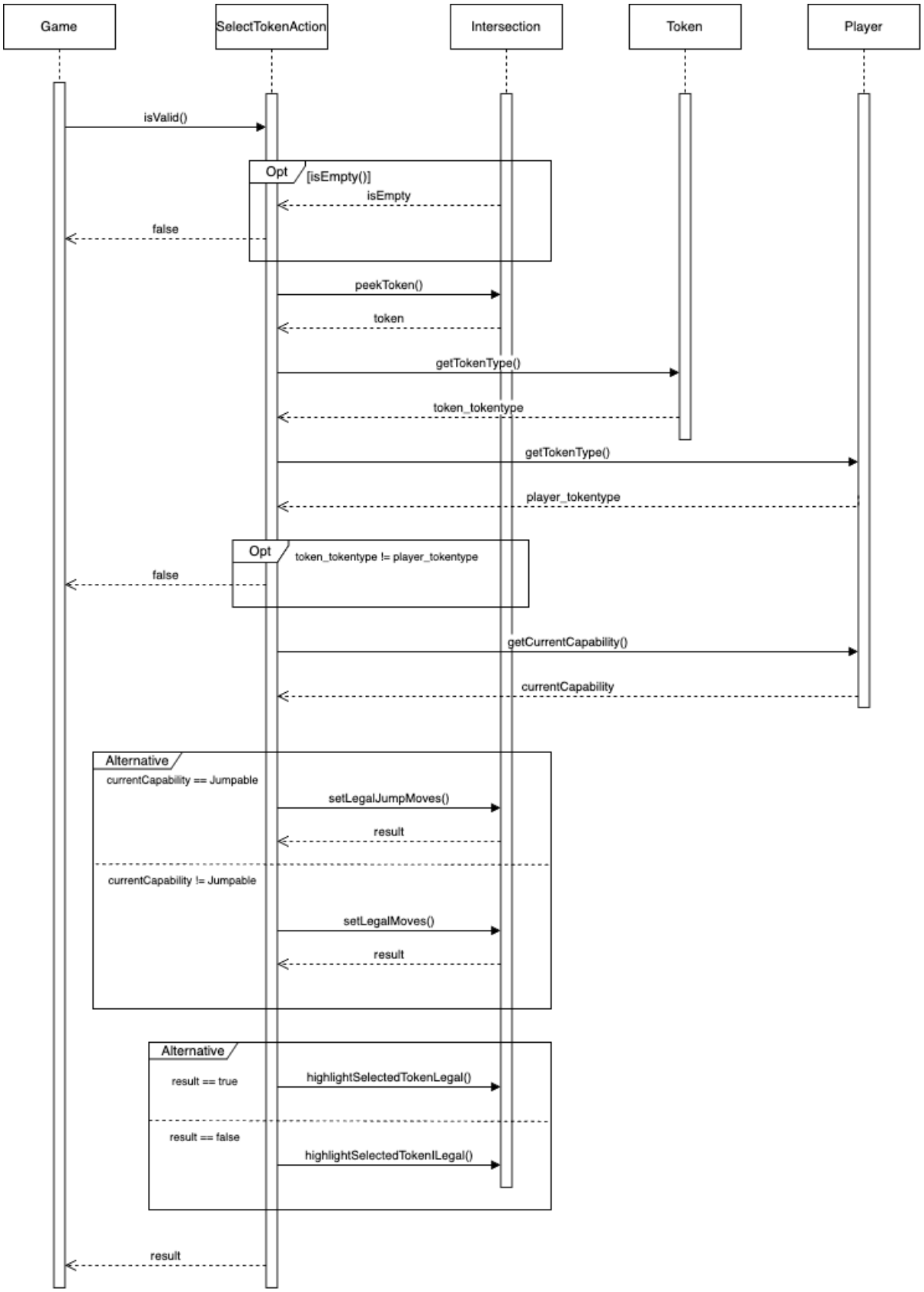
Game Winning



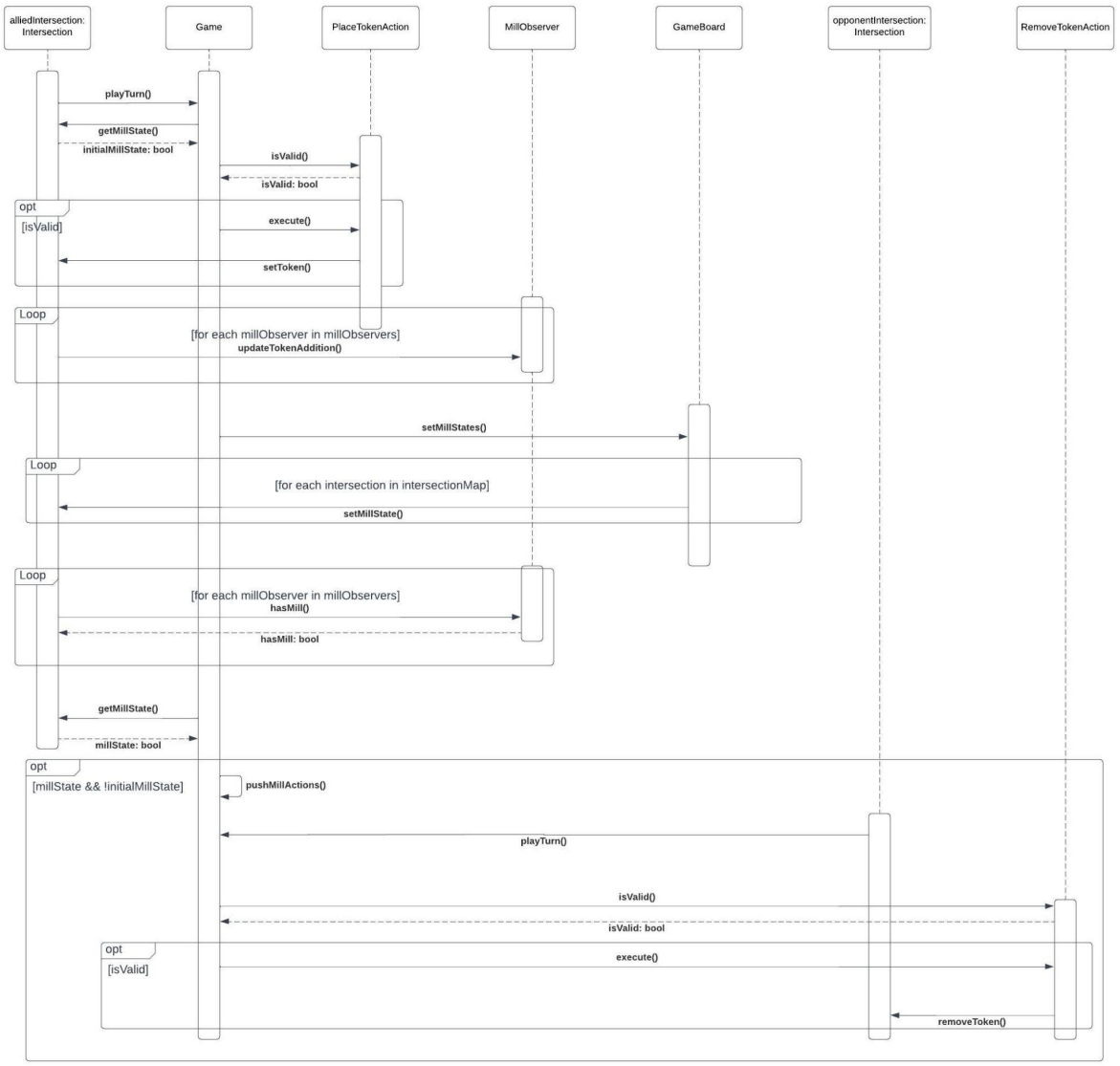
Token Movement



Token Selection Validation + Hinting



Mill Detection + Token Removal



Appendix A: Design Rationale Guidelines

Guidelines

Guidelines for Writing Good Design Rationales

- What is the Decision Point (what are you trying to decide)?
- What are the design alternatives/options?
- What are the advantages and disadvantages of each?
- What is the final decision based on above considerations?

An Example

Issue: A single player and a multiplayer are very similar classes. Should we create an abstract class, Player, that encompasses the common features?

Alternative A: Create a Player abstract class.

Advantages:

- Promotes reuse
- Makes the solution easy to extend should there be some other type of gameboard in the future.
- May contribute to simpler algorithms via polymorphism.

Disadvantages: Adds a layer of abstraction to the design.

Alternative B: No Player abstract class.

Advantages: Simpler design.

Disadvantages:

- May not promote the right reuse
- May have to write similar code for common features in two places.
- More work to extend in the future to accept different board type.

Decision: Alternative A. The advantages of A outweigh the disadvantages, while Alternative B has more disadvantages than advantages.

Template

Issue:

Alternative A:

Alternative B:

Advantages: **Disadvantages:**	**Advantages:** **Disadvantages:**

Decision:

Appendix B: Non-Functional and Human Value Template

Quality Attribute:

Considered In Design:

Importance:

Evidence: