FIT 3077

Semester 1

Architecture and Design Relation

Torino Development United
Soo Guan Yin, Chua Jun Jie, Justin Chuah, Lim Fluoryynx

Table of Contents

Content for Sprint 3 is from Change Log (page 23) and below.

YouTube Link

This is the link to our demo for the game.
https://www.youtube.com/watch?v=kR9K8Zcwcdw

Design Rationale

Key classes that were debated

Node class

Node was initially planned to exist as attributes in the Engine class with the logic that the board, which is represented using the Engine class, consists of nodes. However, we realised that separating Node as a class rather than having it as an attribute in the Engine class will be more beneficial when developing the game.

By separating Node as a class, we can avoid making Engine class a "God" class as separating Node as a class helps with the separation of concerns, which also obeys the Single Responsibility Principle (SRP) as the code can be more focused on each of their individual components.

Methods and properties of the Node can be defined to its functionality specifically. For example, our Node class can contain a method to check whether the node is occupied by a token or which nodes are adjacent to a specified node.

Separating out methods and properties of the Node class separated the functionality of the board and the node. This separation provides more flexibility as any change in implementation of the node functionality will not affect the implementation of the board's functionality or vice versa. It will also be easier to modify the behaviour of the Node class or add new features during future development by modifying the Node class instead of the Engine class.

In addition to that, separation of concerns allows us to reuse the Node class's functionality in other parts of the game, which for our case is implementing the logic of Player's move. When implementing the logic for a player to move their token from one node to another, the logic to determine which nodes are valid moves will be encapsulated in the Node class as methods and we can reuse them in multiple parts of the game without rewriting the logic for each use case. This saves time and increases efficiency in the development process while ensuring code is consistent and maintainable.

Action class

Action was initially planned to be methods in the Actor class with the logic that the actor performs action. However, we ended up separating Action as a class instead.

Similar to the Node class explanation above, separating Action as a class helps with the separation of concerns, which also helps in implementing the Single Responsibility Principle

(SRP) as the code can be more focused on each of their individual components without mixing responsibilities.

By doing so, Actor class will be responsible for storing player's data such as colour, number of tokens, status etc; while Action class will be responsible for storing data related to moves and representing specific actions made by players during the game, such as putting a token on the board or removing an opponent's token.

Furthermore, the game can be modified without affecting the Actor class, and vice versa. If we were to change the game rules or add new actions, we can make changes to the Action class without making changes to the existing Actor class. We can easily create new Action subclasses to handle new actions without making changes to the Actor class. This will make our code more maintainable and extensible in the future.

Separating Action as a class instead of making them as methods in the Actor class also helps with encapsulation. The data and functionality of Actor's action will be encapsulated within Action class. This reduces coupling between Actor class and the game logic. We can ensure that the Actor class can only access data and methods for it to perform its own responsibilities, hence making the code more robust and maintainable.

Actions players can perform during a game will not be appropriate to be methods because if an Actor has many types of actions, it can perform such as putting a token, moving a token, removing an opponent's token etc, all these methods inside the Actor class will make it inconvenient to add new actions or modify existing ones.

Key Relationship in Class Diagram

Relation Between Node and Token

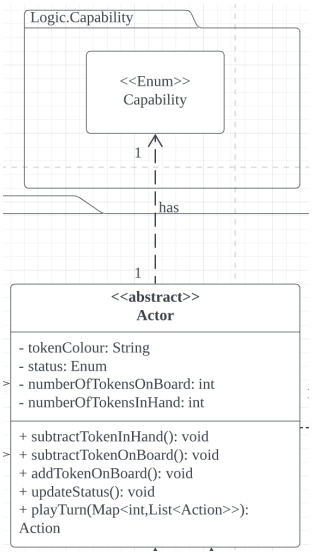
The Node class has an association with the Token class, as in each cycle, the Board class must examine each node in the list to determine all legal moves that a player may make. To determine whether any legal moves exist on a given node, the node must check if any tokens are present on it. Therefore, if a token occupies a node, the token's information must be stored in the node so that it can examine what legal moves a player can make at that location.

Relation Between Node and Action

The Node class is an essential component of the game logic that determines the available moves a player can make at any point in the game. To achieve this, the Node class relies on the Action class, which provides a list of actions a player can take when their token is on a specific node. By utilizing this dependency, the Node class can generate the available actions dynamically, based on the presence of tokens on the current node and its neighbouring nodes, without storing all the possible actions for each node redundantly. This approach saves memory and increases game speed while ensuring that players have access to up-to-date actions.

Multiplicities rationale

Actor and Capability

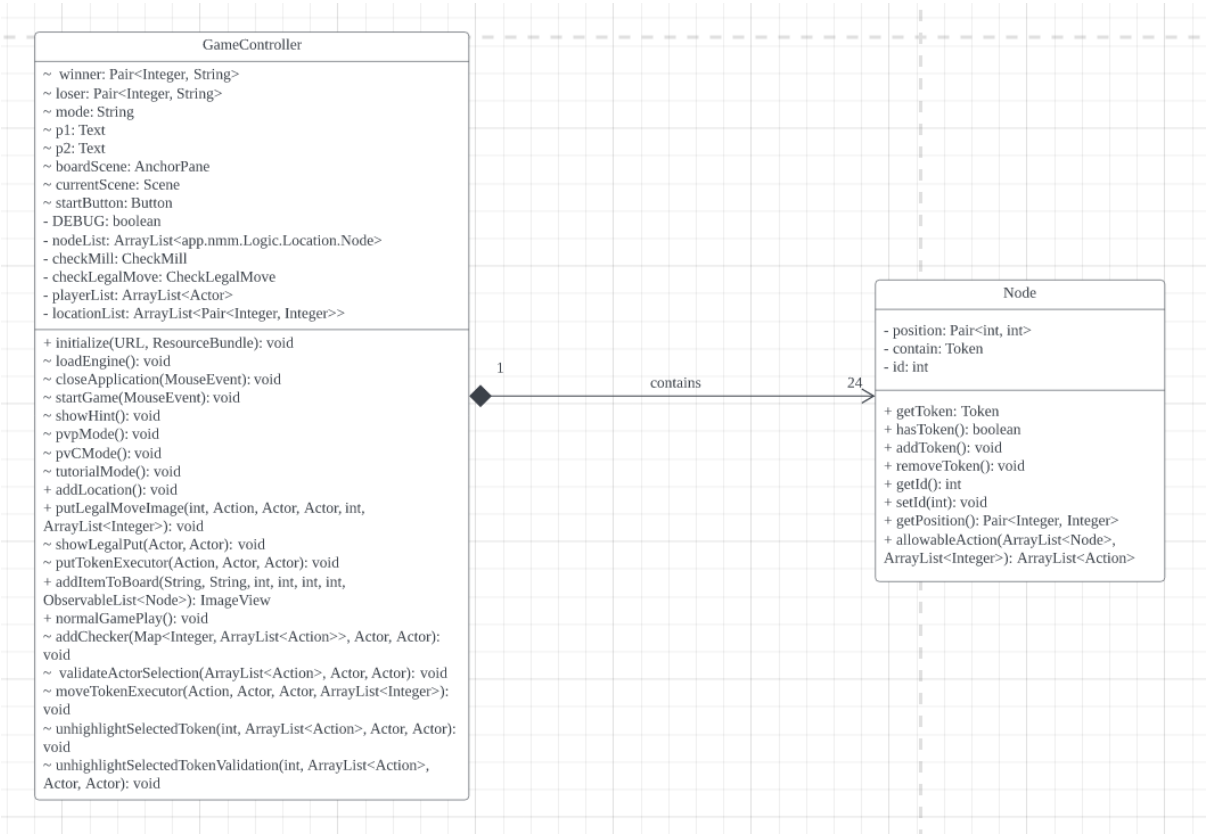


According to the multiplicity shown above, one actor can have one capability (from the Capability Enum class) at a time. The multiplicity between the Actor class and Capability class is represented as 1 -> 1 because Actors will be given capabilities based on the number of tokens they have on the board, and number of tokens they have with them (not placed on the board yet), as shown in table below.

Capability	**Condition**
PUT_TOKEN	When player still have >0 tokens with them
MOVE_TOKEN	When player have 0 tokens with them
FLY_TOKEN	When player is left with 3 tokens on the board and 0 tokens with them

Since removed tokens will not reappear in the game, the total number of tokens left with Actor will only decrease overtime. Actor's capability will be replaced with a new capability when their condition changes. Therefore, Actors will only have either one of the capabilities from the table above hence the number of capabilities an Actor can have is 1.

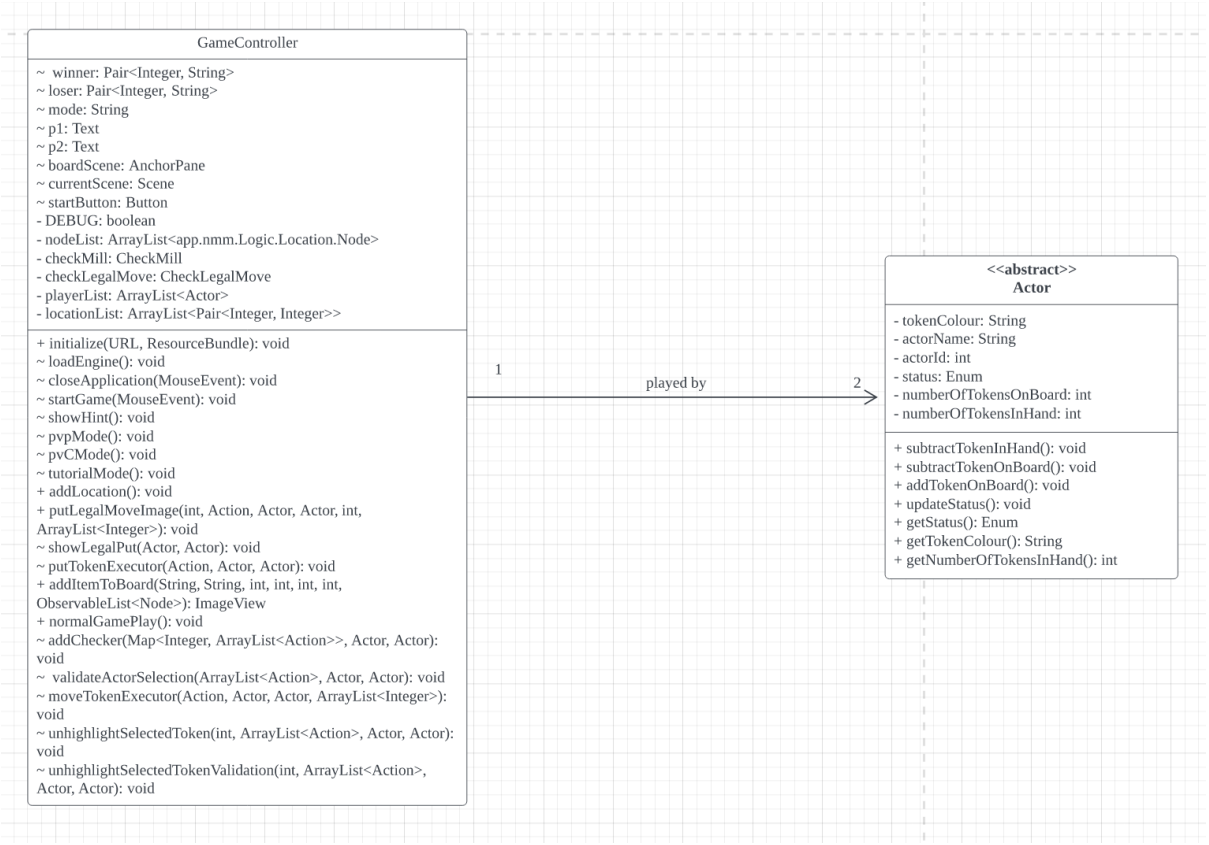
GameController and Node



GameController class and Node class have a multiplicity of 1 -> 24 because the number of nodes remains the same throughout the game, which is 24 Nodes.

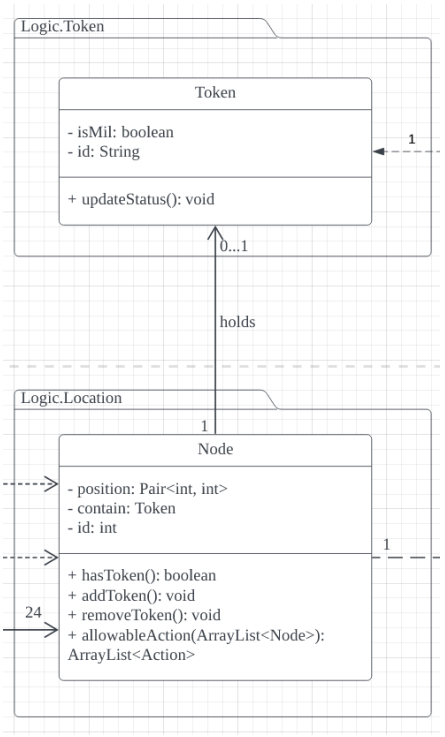
In a Nine Man Morris game, the board consists of 24 intersections, which are nodes connected by straight lines, with each Node connecting to 1- 4 neighbouring nodes depending on its position. Therefore, the GameController class will have 24 instances of the Node class in the constructor, represented by the nodeList attribute which includes an array list containing the 24 nodes for tokens to be placed in the game.

GameController and Actor



GameController class and Actor class have a multiplicity of 1-> 2 because each instance of GameController is played by two Actors (the game will not start if there is less than two Actor). GameController class represents the current game state while the Actor class represents players that are participating in the game. Since there are always 2 Actors participating in a game, there must always be two instances of the Actor class that each instance of the GameController class have association to as the GameController class have two instances of Actor in the playerList attribute, which is a list containing two Actor involved in the game.

Node and Token



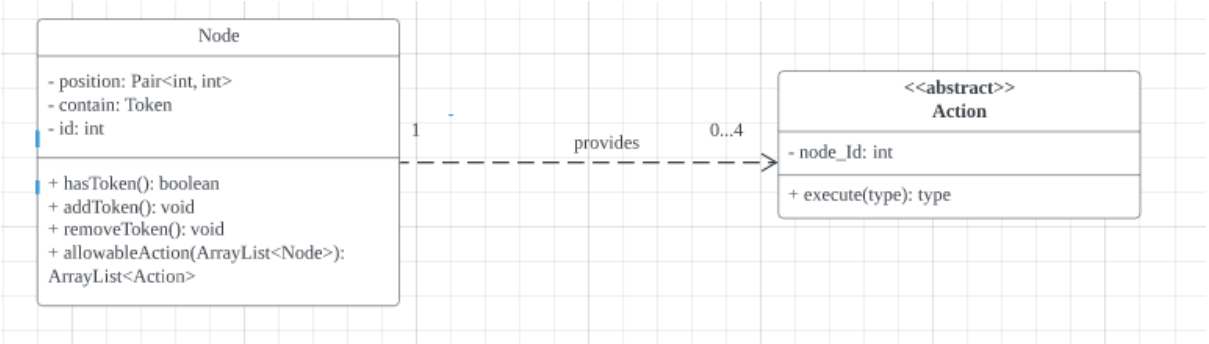
The multiplicity between Node class and Token class is 1 -> 0…1 because each Node on the Board may or may not have a Token placed on it.

Node class represents individual intersections on the Board while Token represents tokens that are placed on the nodes by players.

The multiplicity between Node and Token class is 1 -> 0…1 because not all nodes on the board will necessarily have a token being placed on them at any given time.

0 indicates no token on the node while 1 indicates that there is a token on the node. In other words, each instance of the Node class can only be associated with either zero or one Token class. The ‘contain’ attribute in the Node class will be a Token class if there is a Token placed on it (1) , and will be null if there is no Token on it (0).

Node and Action



The multiplicity between Node class and Action class is 1 -> 0…4 because a Node can give an Actor up to 4 allowable actions to be performed. In other words, the allowableAction method returns 0 to 4 instances of Action.

The minimum number of allowable actions will be 0, which is if the Node is occupied by the opponent's token hence no token can be placed on that node and Node will not give the Actor action to be performed.

Since a Node can have up to 4 adjacent nodes, the maximum number of allowable actions will be 4, that occurs if Actor has no tokens left on hand and more than three tokens on the board, and the selected node(with that Actor's token on it) has 4 empty adjacent nodes. In this case there will be 4 allowable actions provided (4 new instances of MoveTokenAction) so that the player can choose to move the token to either one of the empty adjacent nodes.

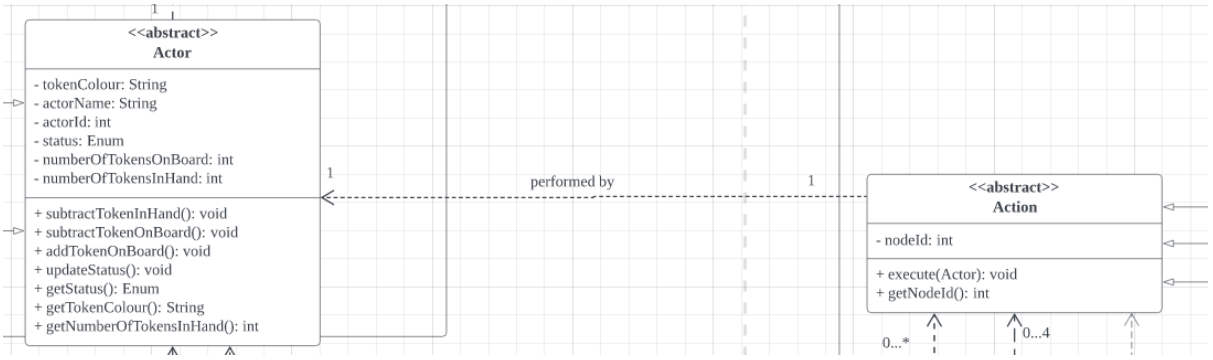
This is because our MoveTokenAction constructor will have the id of empty adjacent nodes the Actor can move to.

```java
public List<Action> allowableAction(List<Node> nodeList, ArrayList<Integer> adjacentList) {
    List<Action> actionList = new ArrayList<>();

    if (this.contain==null) { // if the node is empty
        actionList.add( new PutTokenAction(this.id)); // add a put token action into the list
    }

    else{
        for (int i = 0; i < adjacentList.size(); i++) { // check it's adjacent nodes
            if (nodeList.get(adjacentList.get(i)).contain==null){ // if the adjacent node is empty

                // create and add a move token action towards that empty adjacent node into the list
                actionList.add(new MoveTokenAction(this.id, adjacentList.get(i)));
            }
        }
    }
    return actionList;
}
```

According to the code snippet above, if the given node has 4 empty adjacent nodes, the Node will create 4 new instances of MoveTokenAction. Unlike the PutTokenAction that does not have the id of any Nodes in the constructor, since the PutTokenAction class will have its method to only allow an Actor to put the token on any selected Node if that Node is not occupied by a token.

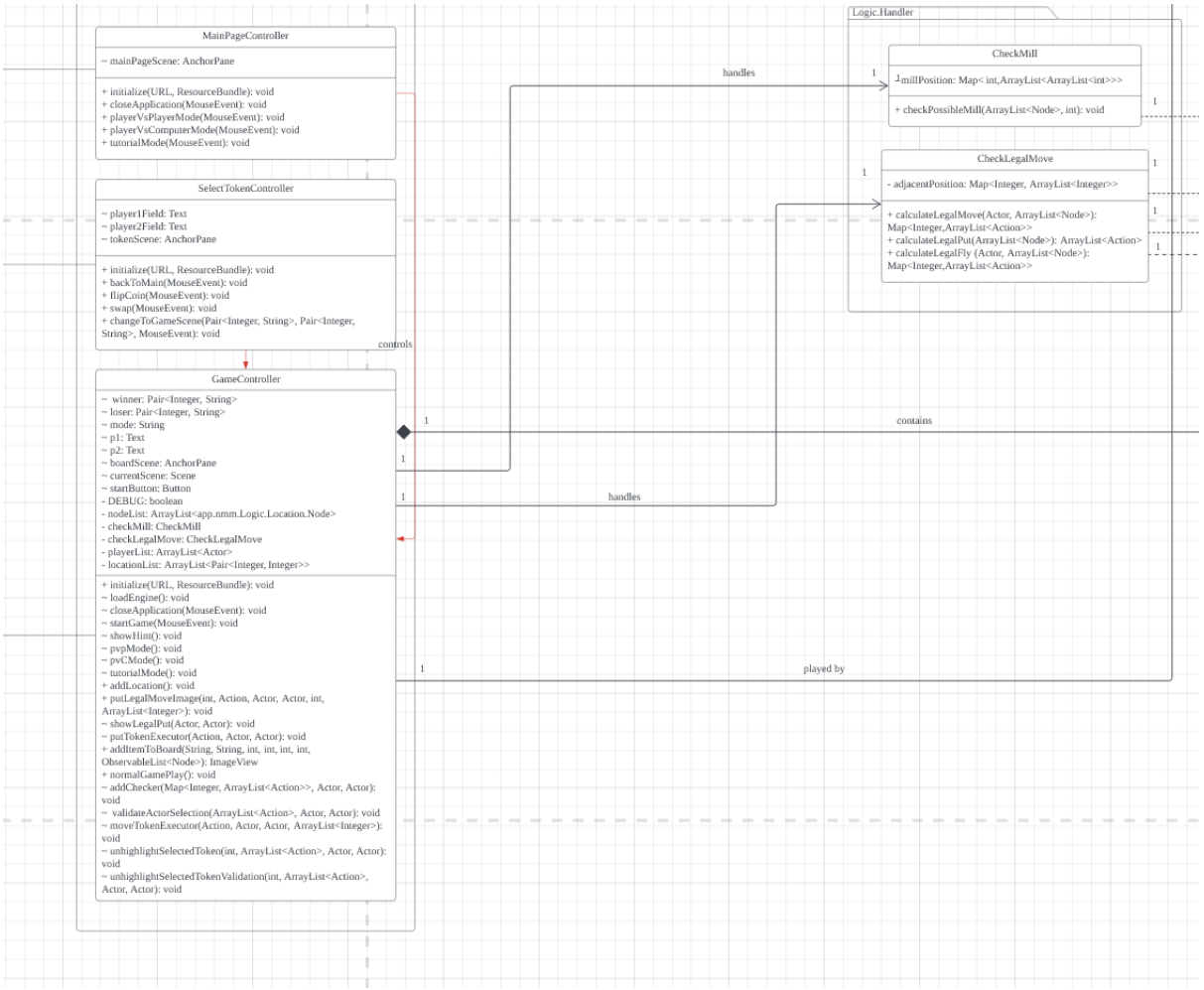
Hence the allowableAction method in the Node class can return a minimum of 0 Action and a maximum of 4 Action instances in the list.

Action and Actor



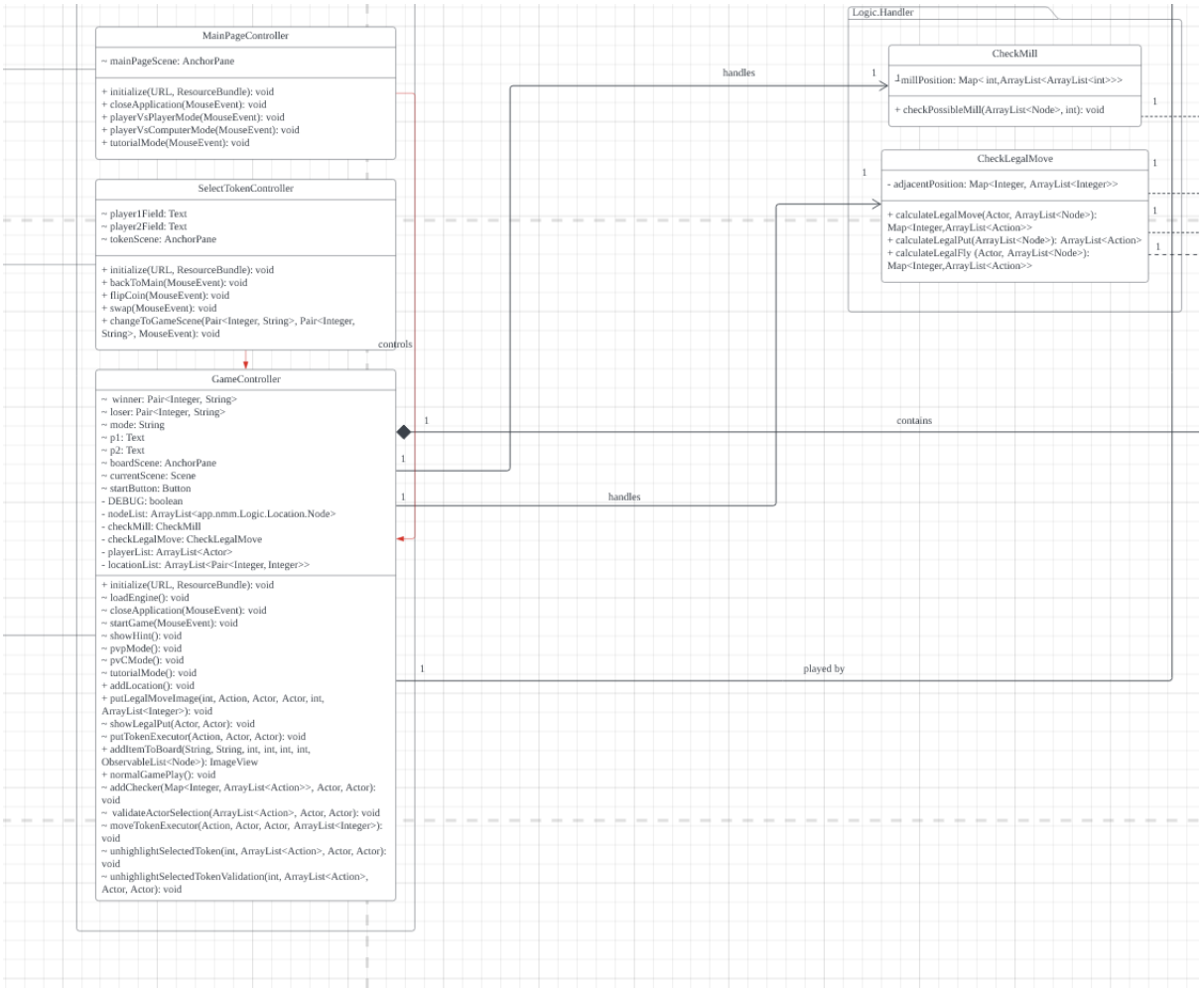
The multiplicity between Action and Actor is 1 -> 1 because the execute method in the Action class references one instance of Actor in the method parameter. In other words, an Action can only be performed by one Actor at a time. The execute method in the Action class changes the attributes in the Actor class, such as number of tokens remaining and capability at a time. Since the execute method will only manage one Actor at a time, the multiplicity of Action to Actor class is 1 -> 1.

GameController to CheckMill



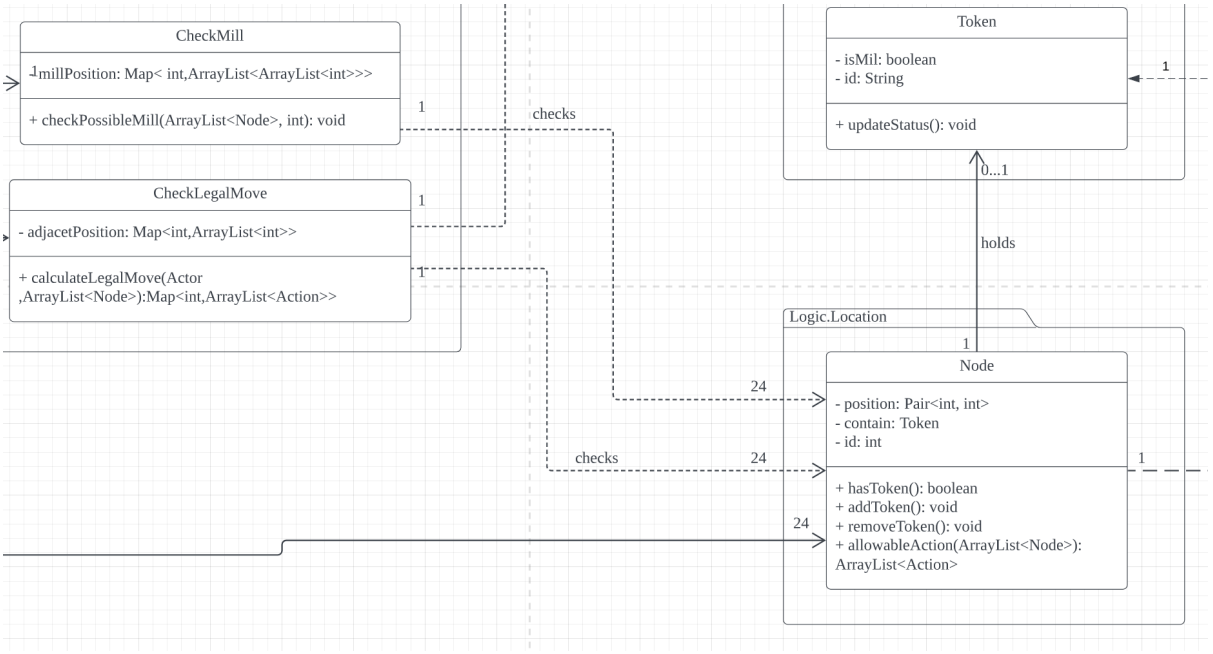
The multiplicity from GameController to CheckMill class is 1 -> 1 because there will only be one instance of CheckMill in the constructor of the GameController class, which is represented by the checkMill attribute. The CheckMill class acts as a “mill manager” during a game play that contains a list of combinations of node ids that can form a mill and method to check whether a mill is formed by Actor. Hence only one instance of the CheckMill class is needed during the gameplay.

GameController to CheckLegalMove



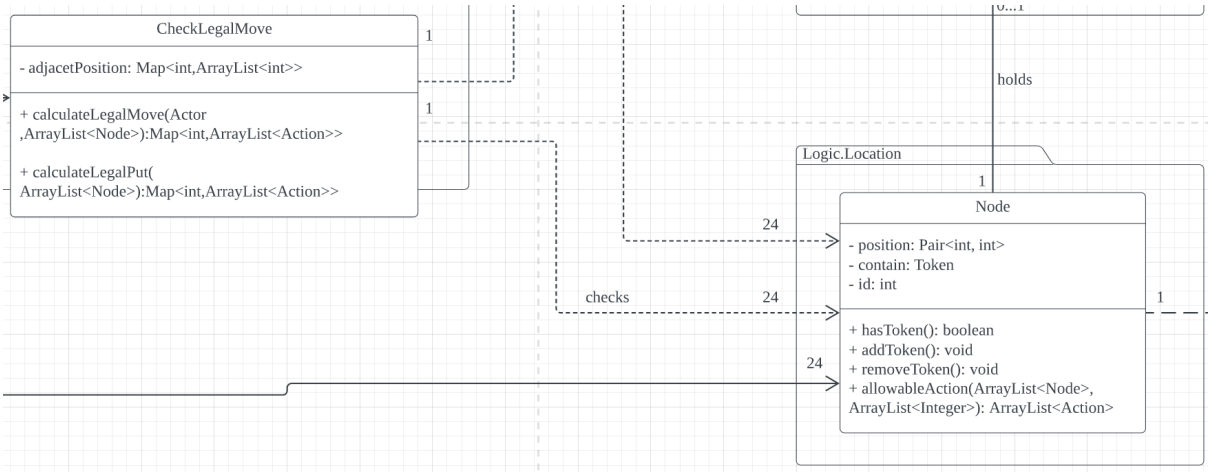
The multiplicity from GameController to CheckLegalMove class is 1 -> 1 because there will only be one instance of CheckLegalMove in the constructor of the GameController class, which is represented by the checkLegal attribute. The CheckLegalMove manages moves that Actors are allowed to perform during their play turn. Hence only one instance of the CheckLegalMove class is needed during a game play.

CheckMill and Node



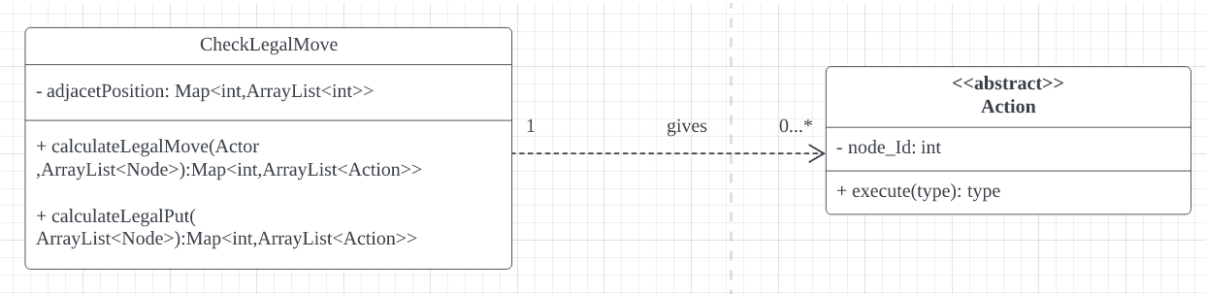
The multiplicity from CheckMill to Node class is 1 -> 24 because the CheckMill class reference 24 instances of Node class (ArrayList<Node>) is the list containing all 24 nodes in the game), which is passed from the Engine class as parameters, in the checkPossibleMill method.

CheckLegalMove and Node



The multiplicity from CheckLegalMove to Node class is 1 -> 24 because the CheckLegalMove class reference 24 instances of Node class (ArrayList<Node> is the list containing all 24 nodes in the game), which is passed from the Engine class as parameters, in the checkLegalMove and the checkLegalPut method. Since only one of the two methods will be called at a time, means a maximum of 24 instances of Node will be referenced each time hence the multiplicity 1 -> 24.

CheckLegalMove and Action



The multiplicity from CheckLegalMove to Action class is 1 -> 0…* because the calculateLegalMove and calculateLegalPut method in the CheckLegalMove class will return 0 to n instances of Action. Since only one of the two methods will be called at a time, 0 instances of Action will be returned if all tokens of the Actor (with no more tokens left" in hand") on the board are surrounded by tokens (no empty adjacent nodes for the token to be moved). While the upper bound is represented by * as the number of Action instances returned may vary depending on the status of both players and/or arrangement of tokens on the board during the game.

Design Patterns

Chosen Design Pattern to Explain (Factory Method)

One of the design patterns that we have applied for the construction of our class diagram is the Factory Method design pattern. We created our Abstract Action class with thorough consideration of the Factory Method. We already expect our Action class to not really require any extensions of possible actions since the gamerules of Nine Men Morris can be considered to be straightforward, but even if there is a need for extra functionalities, an extra subclass can just be extended to the Abstract Action class.

With our current design of calibrating all the possible legal moves that can be made by the Players(and Computer), we figured that it is best to not overcomplicate this part of the system and take a relatively simple approach to check the possible legal actions that can be made by the Players(and Computer). As such, we have delegated the responsibility of creating the action objects to the subclasses that extend from the Action abstract class, and then have the objects created by the subclasses to be added directly into a list of capable actions that can be taken by the Player to ensure that every possible action that can be done follows the game rules.

Feasible Alternatives that were discarded

Strategy Design Pattern

This design pattern was originally our initial choice for the design of the Abstract Action class, but it was then discarded since we believe that this approach would only cause complications for our implementation of the Nine Men Morris gameplay. This is because the set of possible actions that can be made by the players are already being limited by the gamerules, allowing us to come up with fairly simple algorithms for the implementation of said actions. Additionally, seeing as to how we also do not require the algorithm to be switched from one another in runtime, and how our Computer's algorithm is merely a simple select randomly from a list of possible legal actions, meaning that there is no need for there to be a priority ranking of which actions would be the most "smartest" action that can be taken by the Computer, we have decided to discard this design pattern and use a more simple approach instead.

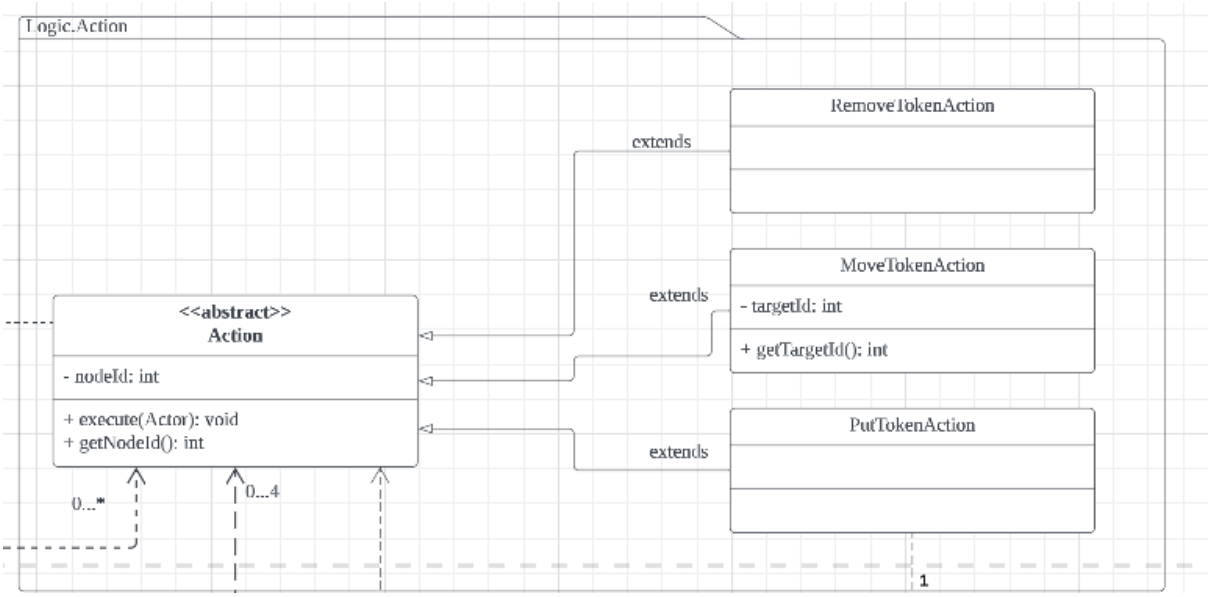
Template Design Pattern

A long discussion was conducted within the team to determine whether a template design pattern should be used or a factory design pattern for the design of the abstract Action class. However, after thorough discussion, we have understood that the subclasses that extend from the abstract Action class only change the implementation of one method and for certain subclasses, the execute parameters are different from one another. This means that if we were

to design it based on the template design pattern, it would be unnecessary to overcomplicate the system since only one specific method is to be changed, hence causing us to just delegate the responsibility of the object creation directly to the subclasses by using the factory method design pattern instead.

Inheritance

Action Abstract Class

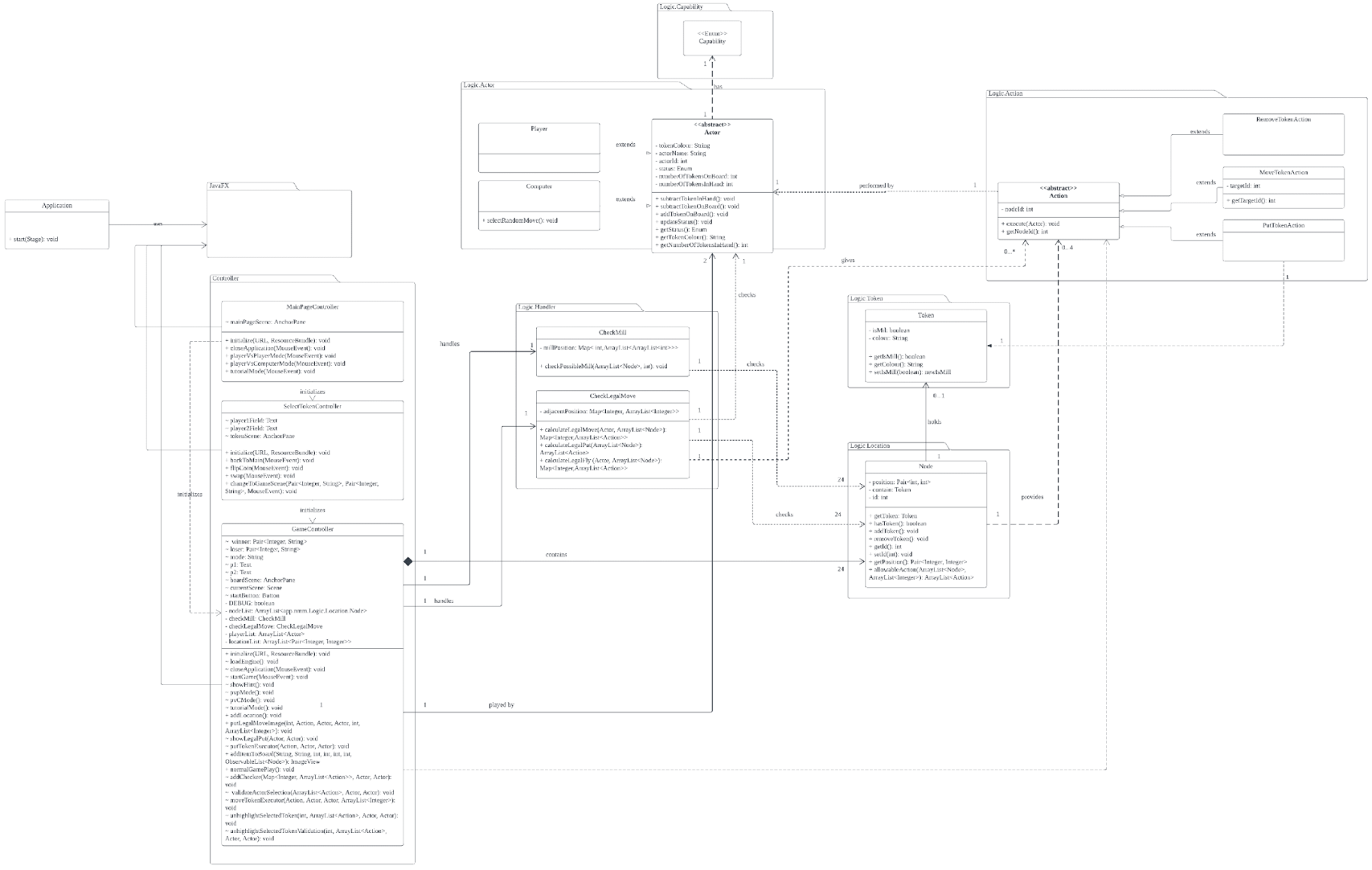


The updated version of the Action Abstract class from the last sprint.

We have decided to keep using an Abstract class for Actions and have the RemoveTokenAction, MoveTokenAction and PutTokenAction extend from it.

The reasoning behind this is the same as in the previous Sprint which is to ensure that the DRY principle is not violated since the classes have commonly used methods which would end up being repeated if they do not use inheritance. And since we want to adhere to the Single Responsibility Principle as well, we chose not to use conditionals and grouped them into one class, but instead broke them down into three separate classes. Since by doing this, it also allows us to adhere to the Open Closed Principle, it can be considered to be safe for future implementations if there ever calls for a need to extend the algorithm of the Computer move selection.

Class Diagram


Logic.Capability
<<Enum>>
Capability
Logic.Actor
has
Player
Computer
+ selectRandomMove(): void
extends
<<abstract>>
Actor
- tokenColour: String
- actorName: String
- actorId: int
- status: Enum
- numberOfTokensOnBoard: int
- numberOfTokensInHand: int
+ subtractTokenInHand(): void
+ subtractTokenOnBoard(): void
+ addTokenOnBoard(): void
+ updateStatus(): void
+ getStatus(): Enum
+ getTokenColour(): String
+ getNumberOfTokensInHand(): int
performed by
Logic.Action
<<abstract>>
Action
- nodeId: int
+ execute(Actor): void
+ getNodeId(): int
RemoveTokenAction
MoveTokenAction
- targetId: int
+ getTargetId(): int
PutTokenAction
gives
JavaFX
Application
+ start(Stage): void
uses
Controller
MainPageController
~ mainPageScene: AnchorPane
+ initialize(URL, ResourceBundle): void
+ closeApplication(MouseEvent): void
+ playerVsPlayerMode(MouseEvent): void
+ playerVsComputerMode(MouseEvent): void
+ tutorialMode(MouseEvent): void
initializes
SelectTokenController
~ player1Field: Text
~ player2Field: Text
~ tokenScene: AnchorPane
+ initialize(URL, ResourceBundle): void
+ backToMain(MouseEvent): void
+ flipCoin(MouseEvent): void
+ swap(MouseEvent): void
+ changeToGameScene(Pair<Integer, String>, Pair<Integer, String>, MouseEvent): void
initializes
GameController
~ winner: Pair<Integer, String>
~ loser: Pair<Integer, String>
~ mode: String
~ p1: Text
~ p2: Text
~ boardScene: AnchorPane
~ currentScene: Scene
~ startButton: Button
~ DEBUG: boolean
~ nodeList: ArrayList<gpp.mem.Logic.Location.Node>
~ checkMill: CheckMill
~ checkLegalMove: CheckLegalMove
~ playerList: ArrayList<Actor>
~ locationList: ArrayList<Pair<Integer, Integer>>
+ initialize(URL, ResourceBundle): void
~ loadEngine(): void
~ closeApplication(MouseEvent): void
~ startGame(MouseEvent): void
~ showHint(): void
~ popMode(): void
~ pvCMode(): void
~ tutorialMode(): void
+ addLocation(): void
+ putLegalMoveImage(int, Action, Actor, Actor, int, ArrayList<Integer>): void
~ showLegalPut(Actor, Actor): void
~ putTokenExecutor(Action, Actor, Actor): void
+ addItemToBoard(String, String, int, int, int, int, ObservableList<Node>): ImageView
~ normalGamePlay(): void
~ addCheckers(Map<Integer, ArrayList<Action>>, Actor, Actor): void
~ validateActorSelection(ArrayList<Action>, Actor, Actor): void
~ moveTokenExecutor(Action, Actor, Actor, ArrayList<Integer>): void
~ unhighlightSelectedTokes(int, ArrayList<Action>, Actor, Actor): void
~ unhighlightSelectedTokesValidation(int, ArrayList<Action>, Actor, Actor): void
handles
initializes
Logic.Handler
CheckMill
- millPosition: Map< int,ArrayList<ArrayList<int>>>
+ checkPossibleMill(ArrayList<Node>, int): void
CheckLegalMove
+ adjacentPositions: Map<Integer, ArrayList<Integer>>
+ calculateLegalMove(Actor, ArrayList<Node>): Map<Integer,ArrayList<Action>>
+ calculateLegalPut(ArrayList<Node>): ArrayList<Action>
+ calculateLegalFly (Actor, ArrayList<Node>): Map<Integer,ArrayList<Action>>
checks
Logic.Token
Token
- isMill: boolean
- colour: String
+ getIsMill(): boolean
+ getColour(): String
+ setIsMill(boolean): isCheckMill
holds
Logic.Location
Node
+ position: Pair<int, int>
- contain: Token
- id: int
+ getToken: Token
+ hasToken(): boolean
+ addToken(): void
+ removeToken(): void
+ getId(): int
+ setId(int): void
+ getPosition(): Pair<Integer, Integer>
+ allowableAction(ArrayList<Node>, ArrayList<Integer>): ArrayList<Action>
provides
contains
played by

Change Log

This section will discuss the details for the changes for certain classes made in Sprint 3 from Sprint 2.

Just for clarification purposes:
If a specific item in the class diagram was change it will be represented in GREEN
If a specific item in the class diagram was removed it will be represented in ~~RED~~
If a specific item in the class diagram is newly added it will be represented in ORANGE

SelectTokenController Class

Changes

Method

changeToGameScene

- Now it shows an overlay at the board scene where it shows the flip coin result.

GameController Class

Changes

Method

MoveTokenExecutor

- Added code to check if the token belongs to a mill. If it is in a mill, the mill has to be broken and the graphic shown on the UI will need to be changed.
- Added code to check if the token after the move had formed a mill. If a mill is formed , the graphic on the UI will update to show that a mill is formed.
- Added code to check if there are removable opponent tokens on the board, the action to remove tokens will be provided to the current player that formed the mill.

PutTokenExecutor

- Added code to check if the token that is being placed had formed a mill. If a mill is formed , the graphic on the UI will update to show that a mill is formed.
- Added code to check if there are removable opponent tokens on the board, the action to remove tokens will be provided to the current player that formed the mill.

backToMain

- Changed the implementation of the method such that it brings the user back to the main menu instead of quitting the application.

normalGameplay

- Added a feature so that the game checks if the current player has any legal moves before proceeding to the turn-based gameplay.

addUnhighlightSelectedToken

- Refactored the name of the method. It is previously known as unhighlightSelectedToken.

unhighlightSelectedToken

- Refactored the name of the method. It is previously known as unhighlightSelectedTokenExecutor.

Newly Added

Attribute

confirmButton and unconfirmedButton

- Added attributes for the pause overlay

clicked and prevNodeId

- Added attributes to facilitate the code for highlighting legal moves

.

windowResourcePath and macResourcePath

- Added attributes to search for the resource folder.

reasonWin

- This text field is added to display the condition of one of the player that made the other player the winner based on their number of legal moves or number of tokens left on board

displayWinner

- This attribute is added as a text field in the end game overlay to display the colour of the winner.

gameStatus

- This attribute is added as a text field to display the player's turn and type of action to be performed during the gameplay.

Method

updateTokenCount

- This method is in charge of updating the number of tokens each player has on board on the UI side. By adding this method, it allows the application to obey the DRY principle as there are more than one method that is required to use the code

swapTokenToMill

- This method is used to swap the graphic of the token on the UI to the appropriate graphic that represents the token as part of a mill. Besides that, it also sets the token to be part of a mill, so that the opponent user will not be able to remove it when he/she forms a mill. By adding this method, it allows the application to obey the DRY principle as there are more than one method that is required to use the code.

changeTokenImage

- This method is in charge of changing the graphic of the node into another graphic with the user specific size and offset. By adding this method, it allows the application to obey the DRY principle as there are more than one method that is required to use the codeBy adding this method, it allows the application to obey the DRY principle as there are more than one method that is required to use the code

endGame

- This method is used to set the text to be displayed in the end game overlay. It displays the winner, reason the other player lost and back to the main menu button. This method is added so that we do not need to repeatedly set text in the end game overlay text fields as there are multiple parts in the GameController class that checks whether

players have legal move or at least three tokens during their turn in order to assign legal moves, hence adhering to the DRY principle.

closeOverlay

- A method to close the pause menu overlay

showBackToMainOverlay

- A method to show the pause menu overlay

backToMain

- This method is added as an on click function to the “back to main” button to bring player back to the main menu when clicked

removeImage

- A method to remove all the images with the matching fx:id on the board.

updateTokenCount

- A method to update the token count in the UI

getTokenImagePath

- There are two methods with the same name but different signatures. Regardless, the purpose of the method is to produce the path of the image in the system.

The following 6 methods are implemented similarly due to the fact that the first 3 methods are for the put token phase and the last 3 methods are for the move token phase. This is to avoid the need to constantly check for certain specific conditionals in each individual method before being able to call the relevant functions, improving the overall readability of the function that calls these methods.
showPutRemovable

- This method is called when there are removable opponent tokens on the board during the put token phase. The game text would be updated and putRemoveTokenImage would then be called to update the UI token images to show the possible removable opponent tokens.

putRemoveTokenImage

- This method is added to place the removable token images in the UI for the relevant removable tokens in the put token phase.

putRemoveTokenExecutor

- This method is created to allow the current player to interact with the removable token images presented in the UI if the current player had formed a mill during the put token phase.

- When the current player clicks on an opponent token with a removable token image(highlighted in red), the token that was clicked would then be removed from the board and the turn will then be passed.

showMoveRemoval

- This method is called when there are removable opponent tokens on the board during the move token phase. The game text would be updated and moveRemoveTokenImage would then be called to update the UI token images to show the possible removable opponent tokens.

moveRemoveTokenImage

- This method is added to place the removable token images in the UI for the relevant removable tokens in the move token phase.

moveRemoveTokenExecutor

- This method is created to allow the current player to interact with the removable token images presented in the UI if the current player had formed a mill during the move token phase.
- When the current player clicks on an opponent token with a removable token image(highlighted in red), the token that was clicked would then be removed from the board and the turn will then be passed.

Token Class

Changes

Method

getIsMill

- Change the way the application checks if the token is part of a mill.

Newly Added

Attribute

isMillHorizontal

- Added a way to indicate the token is part of horizontal mill

isMillVertical

- Added a way to indicate the token is part of vertical mill

Method

setHorizontalMill

- Added a setter to set the token is part of horizontal mill

setHorizontalMill

- Added a setter to set the token is part of vertical mill

Deleted

Attribute

isMill

- This attribute only store True of False, but it does not indicate the token is part of which mill (Token can be part of horizontal or vertical mill or both)

Method

setIsMill

- Remove the method to set the old isMill as there are new ways to set the token to be part of the mill.

CheckMill class

Changes

Attribute

millPosition

- This attribute is a Map that consists of all the possible mill positions which will be the main way of determining whether the tokens on the board have formed a mill.
- The mill positions have been added into the map in a specific sequence to represent if the mill was formed horizontally or vertically respectively.

Newly Added

Attribute

millNodes

- This attribute is added to store the list of nodes that are part of a mill

Method

checkPossibleMill

- The return type of this method is changed to an ArrayList containing a pair of Boolean to indicate whether a mill is formed vertically and/or horizontally. ‘True’ being the first item in the Array list indicates a horizontal mill is formed while ‘True’ being the second item indicates a vertical mill is formed.
- This makes it more convenient to determine which tokens(that are part of a mill) to highlight in the GameController class. It is also used to check whether a player formed a mill during their turn by checking whether the returned array contains ‘True’.

CheckLegalMove class

Newly Added

Attribute

currentActions

- This attribute is created to store allowable actions of the current player which would then be used to display the possible legal moves in the UI

currentRemovables

- This attribute is created to store an array of removeActions that can be performed by the player during a given turn.
- The actions in this array would be used to determine whether there are removable tokens in the board for the UI.

calculateLegalRemove

- Similar to the calculateLegalMove method, this method is used as a 'setter' for the currentRemovable attribute. This method iterates the list of nodes and checks whether the node contains an opponent's token that is not part of a mill. If so, a RemoveTokenAction on that node will be added into the currentRemovebles attribute.
- The reason why we had separated this method from calculateLegalMove is due to the fact that calculateLegalMove checks for the adjacent positions which complicates the process of determining the possible opponent tokens that can be removed. Since we wanted to avoid using conditionals and decreasing the overall readability for the calculateLegalMove.

getCurrentActions & getCurrentRemovables

- These are just simple getters for currentActions and currentRemovables respectively.

Changes

Method

calculateLegalMove

- this method is changed to a void method instead of having an ArrayList<Action> return type since currentAction is made to be an attribute of the class. Hence the calculateLegalMove method is used as a 'setter' for the currentAction attribute.
- The currentAction will then be accessed using the class's getter method.

Node class

Changes

Method

allowableAction

- Updated the method such that it only processes if the current node has a token or not.

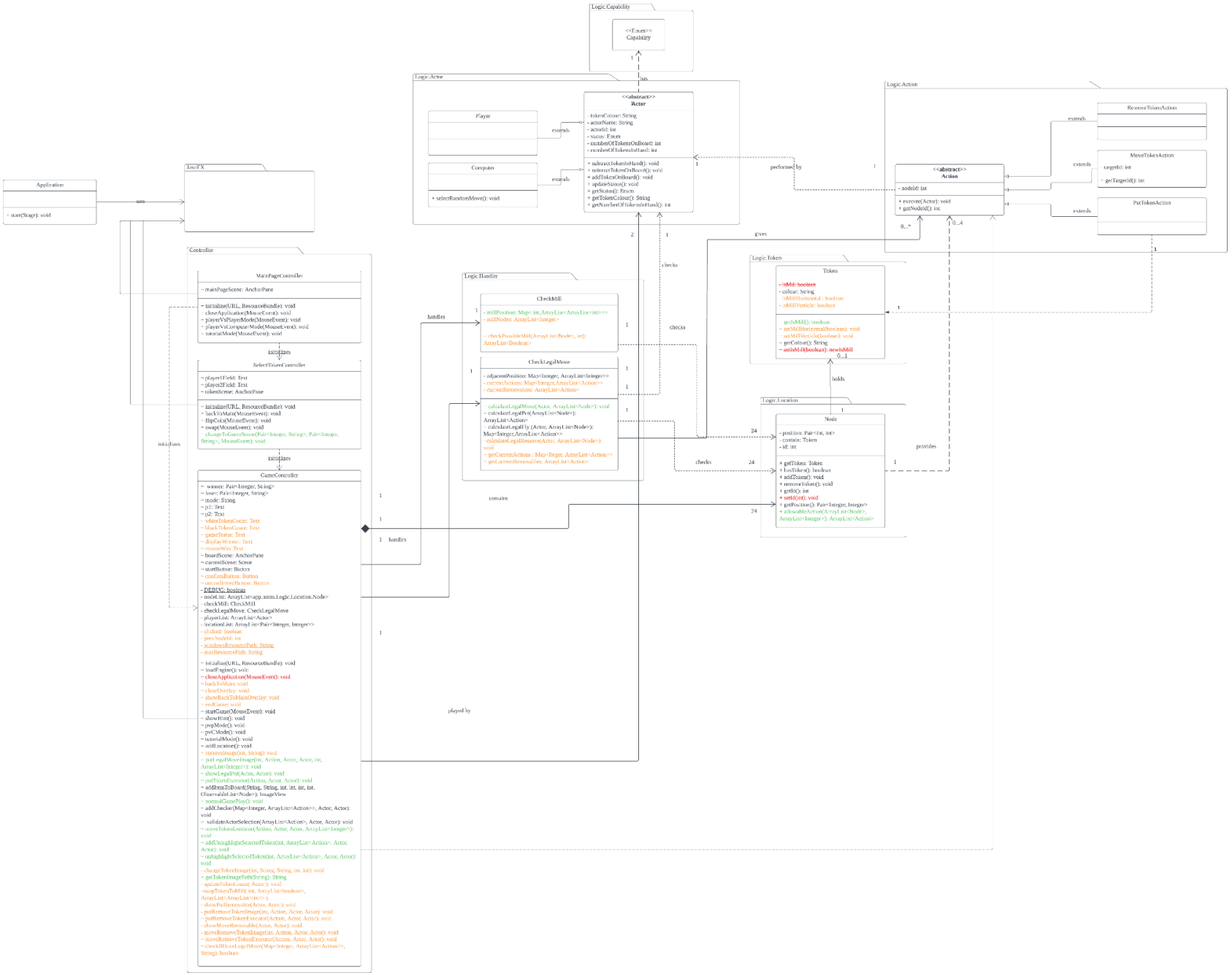
Deleted

Method

setId

- This method was deemed unnecessary since the id of the nodes does not get changed throughout the process of the game.

Revised Class Diagram


Logic.Capability
<<Enum>>
Capability
Logic.Actor
<<abstract>>
Actor
- tokenColour: String
- actorName: String
- actorId: int
- status: Enum
- numberOfTokensOnBoard: int
- numberOfTokensInHand: int
+ subtractTokenInHand(): void
+ subtractTokenOnBoard(): void
+ addTokenOnBoard(): void
+ updateStatus(): void
+ getStatus(): Enum
+ getTokenColour(): String
+ getNumberOfTokensInHand(): int
Player
Computer
+ selectRandomMove(): void
extends
has
performed by
Logic.Action
<<abstract>>
Action
- nodeId: int
+ execute(Actor): void
+ getNodeId(): int
RemoveTokenAction
MoveTokenAction
- targetId: int
- getTargetId(): int
PutTokenAction
gives
0...*
0...4
JavaFX
Application
+ start(Stage): void
uses
Controller
MainPageController
- mainPageScene: AnchorPane
- initialize(URL, ResourceBundle): void
- closeApplication(MouseEvent): void
- playerVsPlayerMode(MouseEvent): void
- playerVsComputerMode(MouseEvent): void
- tutorialMode(MouseEvent): void
initialises
SelectTokenController
~ player1Field: Text
~ player2Field: Text
~ tokenScene: AnchorPane
- initialize(URL, ResourceBundle): void
- backToMain(MouseEvent): void
- flipColor(MouseEvent): void
- swap(MouseEvent): void
- changeToGameScene(Pair<Integer, String>, Pair<Integer, String>, MouseEvent): void
GameController
~ winner: Pair<Integer, String>
~ loser: Pair<Integer, String>
~ mode: String
~ p1: Text
~ p2: Text
- whiteTokenCount: Text
- blackTokenCount: Text
- gameStatus: Text
- displayWinner: Text
- reasonWin: Text
~ boardScene: AnchorPane
~ currentScene: Scene
~ startButton: Button
~ confirmButton: Button
~ unselectRemoveHumanButton: Button
- DEBUG: boolean
- nodeList: ArrayList<app.umm.Logic.Location.Node>
- checkMill: CheckMill
- checkLegalMove: CheckLegalMove
- playerList: ArrayList<Actor>
- locationList: ArrayList<Pair<Integer, Integer>>
- clicked: boolean
- prevNodeId: int
- windowsResourcePath: String
- macResourcePath: String
~ initialize(URL, ResourceBundle): void
~ loadEngine(): void
~ closeApplication(MouseEvent): void
~ backToMain: void
~ closeOverlay: void
~ showBackToMainOverlay: void
~ endGame: void
~ startGame(MouseEvent): void
~ showHint(): void
~ pvpMode(): void
~ pvcMode(): void
~ tutorialMode(): void
~ addLocation(): void
~ removeImage(int, String): void
~ putLegalMoveImage(int, Action, Actor, Actor, int, ArrayList<Integer>): void
~ showLegalPut(Actor, Actor): void
~ putTokenExecutor(Action, Actor, Actor): void
~ addItem2Board(String, String, int, int, int, int, ObservableList<Node>): ImageView
~ normalGamePlay(): void
~ addChecker(Map<Integer, ArrayList<Action>>, Actor, Actor): void
~ validateActorSelection(ArrayList<Action>, Actor, Actor): void
~ moveTokenExecutor(Action, Actor, Actor, ArrayList<Integer>): void
~ addUnhighlightSelectedToken(int, ArrayList<Action>, Actor, Actor): void
~ unhighlightSelectedToken(int, ArrayList<Action>, Actor, Actor): void
- changeTokenImage(int, String, String, int, int): void
- getTokenImagePath(String): String
- updateTokenCount(Actor): void
- swapTokenToMill(int, ArrayList<boolean>, ArrayList<ArrayList<int>>):
- showPutRemovable(Actor, Actor): void
- putRemoveTokenImage(int, Action, Actor, Actor): void
- putRemoveTokenExecutor(Action, Actor, Actor): void
- showMoveRemovable(Actor, Actor): void
- moveRemoveTokenImage(int, Action, Actor, Actor): void
- moveRemoveTokenExecutor(Action, Actor, Actor): void
~ checkIfUserLegalMove(Map<Integer, ArrayList<Action>>, String): boolean
Logic.Handler
CheckMill
- millPosition: Map< int,ArrayList<ArrayList<int>>>
- millNodes: ArrayList<Integer>
- checkPossibleMill(ArrayList<Node>, int): ArrayList<Boolean>
CheckLegalMove
- adjacentPositions: Map<Integer, ArrayList<Integer>>
- currentActions: Map<Integer,ArrayList<Action>>
- currentRemovables: ArrayList<Action>
- calculateLegalMove(Actor, ArrayList<Node>): void
~ calculateLegalPut(ArrayList<Node>): ArrayList<Action>
- calculateLegalFly (Actor, ArrayList<Node>): Map<Integer,ArrayList<Action>>
- calculateLegalRemove(Actor, ArrayList<Node>): void
- getCurrentActions : Map<Integer, ArrayList<Action>>
- getCurrentRemovalTiles: ArrayList<Action>
handles
checks
contains
Logic.Token
Token
- isMill: boolean
- colour: String
- isMillHorizontal : boolean
- isMillVertical: boolean
- getIsMill(): boolean
- setMillHorizontal(boolean): void
- setMillVertical(boolean): void
- getColour(): String
- setIsMill(boolean): newIsMill
holds
Logic.Location
Node
- position: Pair<Int, Int>
- contain: Token
- id: int
+ getToken: Token
+ hasToken(): boolean
+ addToken(): void
+ removeToken(): void
+ getId(): int
+ setId(int): void
+ getPosition(): Pair<Integer, Integer>
+ allowableAction(ArrayList<Node>, ArrayList<Integer>): ArrayList<Action>
provides
checks
24
1
initialises
playerd by

Link to HD version of the diagram:
https://drive.google.com/drive/folders/1sD5OmvjYZx4I9THuMaqCWDAd9sNY1QM0?usp=share_link
https://lucid.app/lucidchart/63bf74f3-c425-4614-8da7-e301139f3680/edit?viewport_loc=-1479%2C5054%2C5390%2C3631%2C2nCKeNV.VUDD&invitationId=inv_140a3647-228a-430a-b41a-f9e51ad70bed

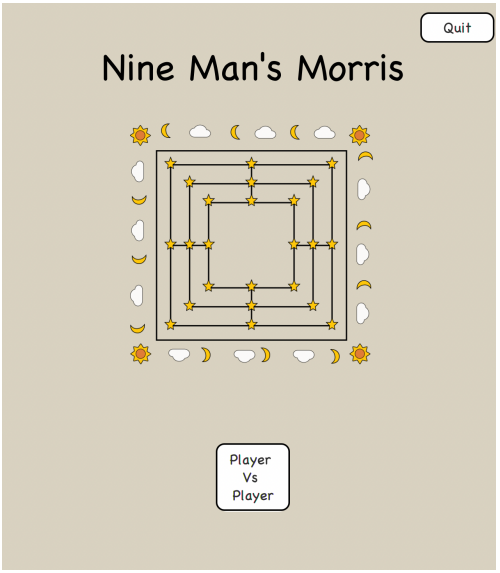
Non-functional requirements

Non-functional requirements are quality attributes or constraints that are unrelated to the functionality of the online Nine Man Morris(NMM) game but are important for the success of the project. Hence the game should be designed such that it meets certain non-functional requirements.
This section discusses the key quality attributes as non-functional requirements that have been explicitly considered in the design of the online Nine Man Morris game by evaluating their relevance to the game and how these quality attributes were manifested in the game design.

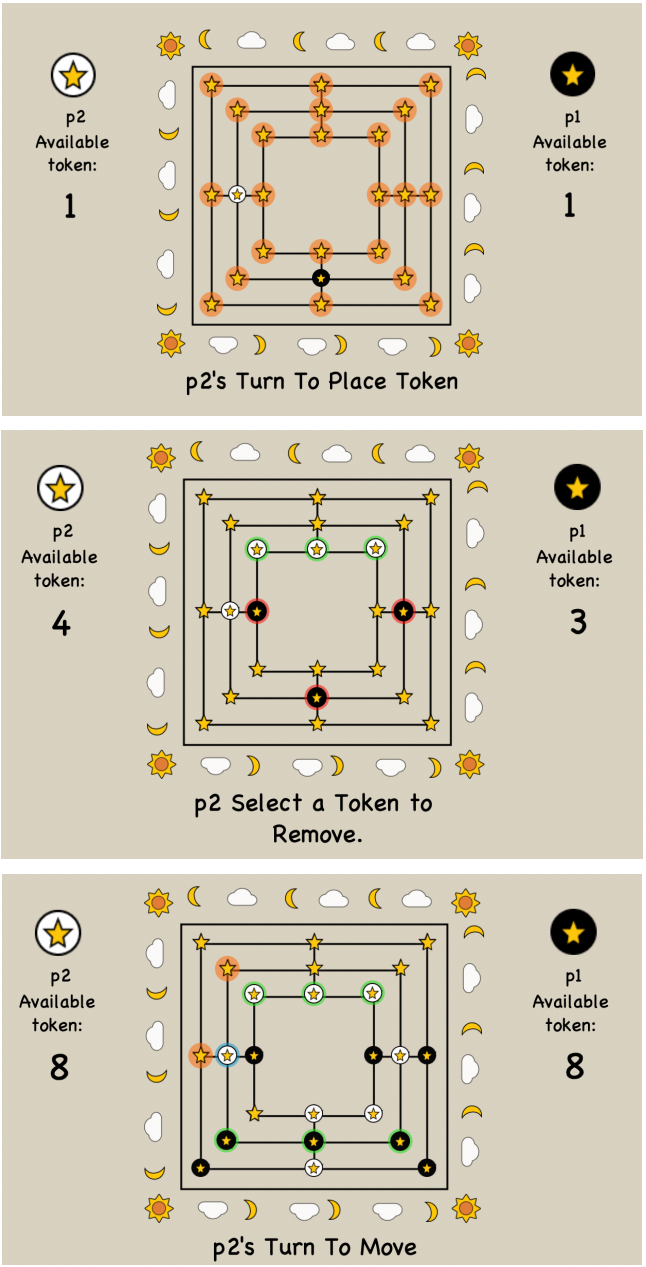
Usability

Usability is important in the online NMM game as it directly affects users' experience and enjoyment of the game. Users may deter from playing it if the game is difficult to use or navigate. In contrast, if the game is easy to use and navigate, it can attract and retain users.
Usability is addressed in the development of the NMM game by considering factors including the ease of accessing the game, clarity of instructions and the ease of performing game action. User testing was conducted to ensure the game's interface is user-friendly and easy to navigate.

1. The game interface is designed such that it is easy to understand and use by only displaying important contents on the screen instead of overpopulating it with irrelevant attributes. The main page only displays important items which are game titles, a simple image that illustrates the game, a button to start the game and a quit button.



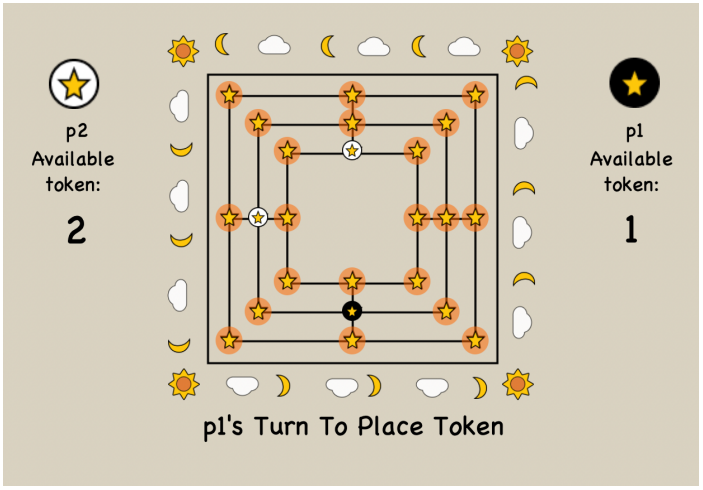
2. Clear instructions and guidelines are provided during the gameplay to help players understand what is happening in the game by displaying the player's turn and number of remaining tokens. When a player's turn is displayed, the instruction also specifies whether the player is supposed to put,move, or remove tokens.



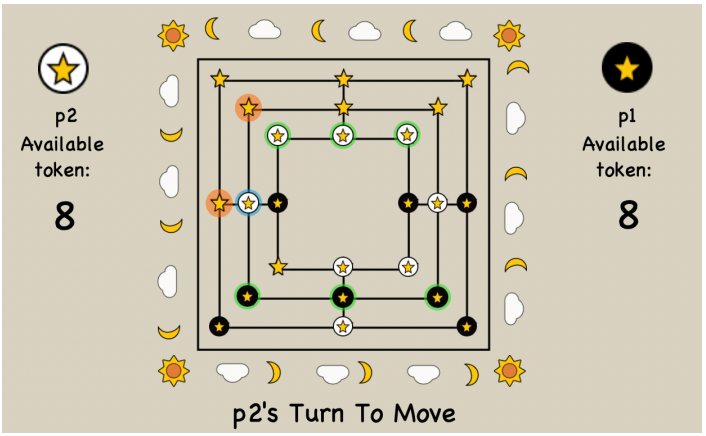
This reduces users' short term memory load as players do not need to manually keep track of their turns and number of tokens.

3. The game is designed using distinct colour schemes for player's to easily differentiate between different elements of the game. Available moves for a selected piece of tokens are shown to players during their turn by highlighting available moves. Tokens that are part of a mill are highlighted in green so that users can clearly see that a mill is formed and tokens highlighted in that colour are not removable.

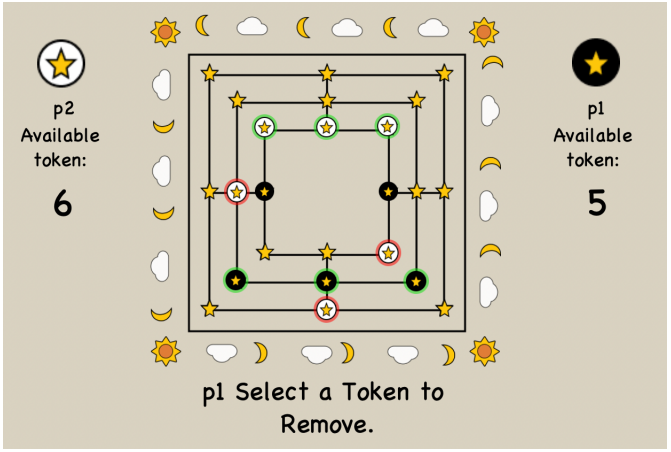
Nodes that a token can be placed to are highlighted in orange.



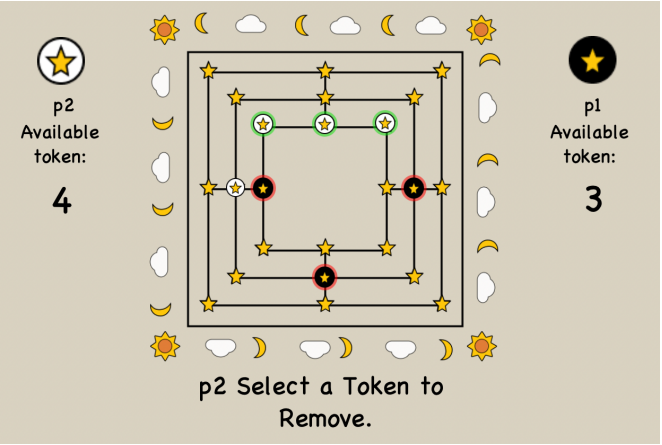
Selected token is highlighted in blue and nodes that the selected token can be moved to are highlighted in orange.



Tokens that are part of a mill are highlighted in green.



Tokens that can be removed are highlighted in red.



By doing so, players will not get distracted from the game's strategic aspects. This significantly saves players' time and improves their ability to strategize during their gameplay, leading to a more satisfying experience.

4. The game provides informative feedback to users, by displaying an overlay announcing the winner when one of the players is left with less than three tokens or no legal moves so that users know why the game ended.

This provides clear communication to players, ensuring no ambiguity about who won the game hence preventing confusion or disputes among players. It also promotes fair play in the game by ensuring all players are aware of the game outcome and that there is transparency in the game's result.

Reliability

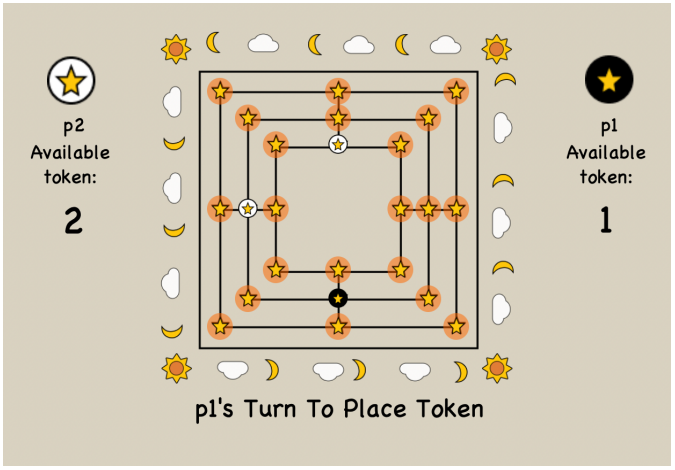
Reliability is another important quality attribute that was considered in the NMM game to ensure the game is consistently available and operates as expected. This involves ensuring the game loads when started and the game adheres to the rules. Ensuring that the game loads when started and adheres to game rules enhances user experience by minimising frustrations as players can start the game without delays or technical difficulties. It also helps to maintain game integrity by ensuring players are under the same set of rules and there are no glitches that give certain players unfair advantages.

To achieve reliability, the game was rigorously tested using various scenarios and analysed the results to ensure game rules are followed with consistent outcomes. This is important for the online NMM game as players expect a fair and consistent experience. Hence the game was thoroughly tested to ensure that it adheres to the game roles and code is bug-free.

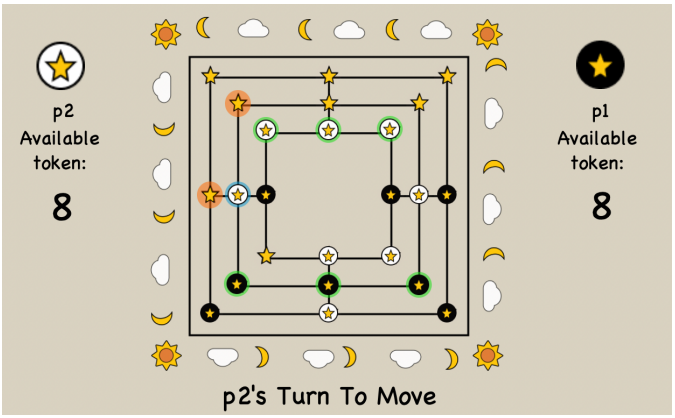
The game will check if a player's move is legal based on the current state of the board, and then highlight legal moves players can perform. To ensure game rules are obeyed, players are only able to move/remove a highlighted token or place a token on a highlighted node during their turn.

Starting the game (game starts when the start button is pressed).

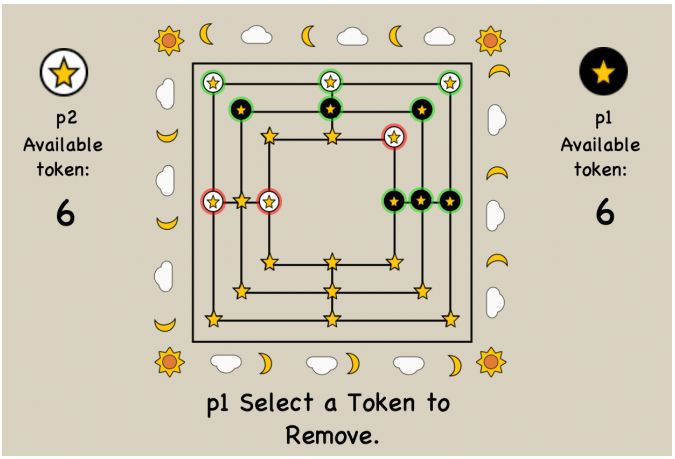
Placing a token on the board (only empty nodes are highlighted).



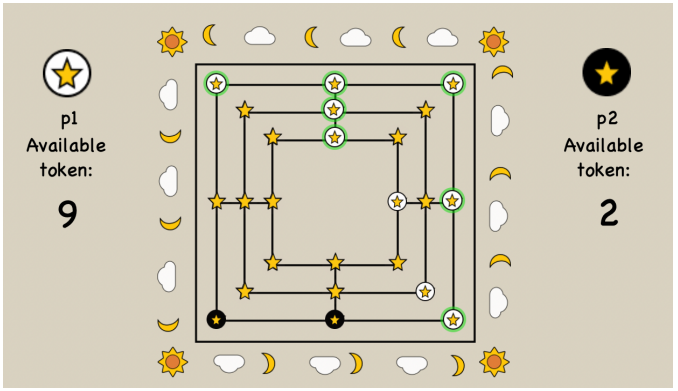
Moving a selected token when a player has more than three tokens on the board (only adjacent nodes are highlighted).



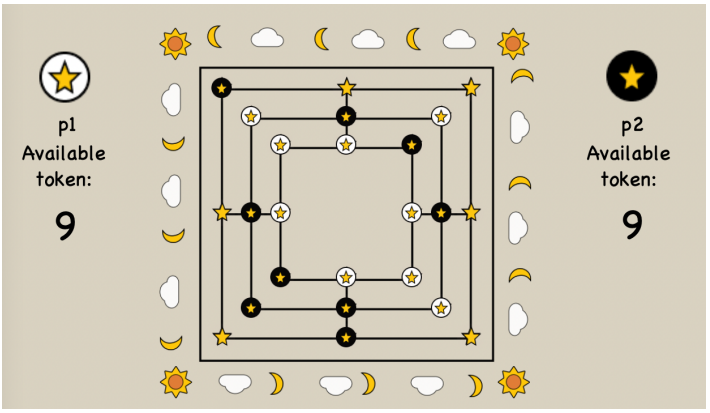
Removing an opponent's token after a mill is formed (only tokens that are not part of a mill are highlighted).



Winning condition 1 (when a player is left with less than three tokens on the board).



Winning condition 2 (when a player has no legal moves to perform).



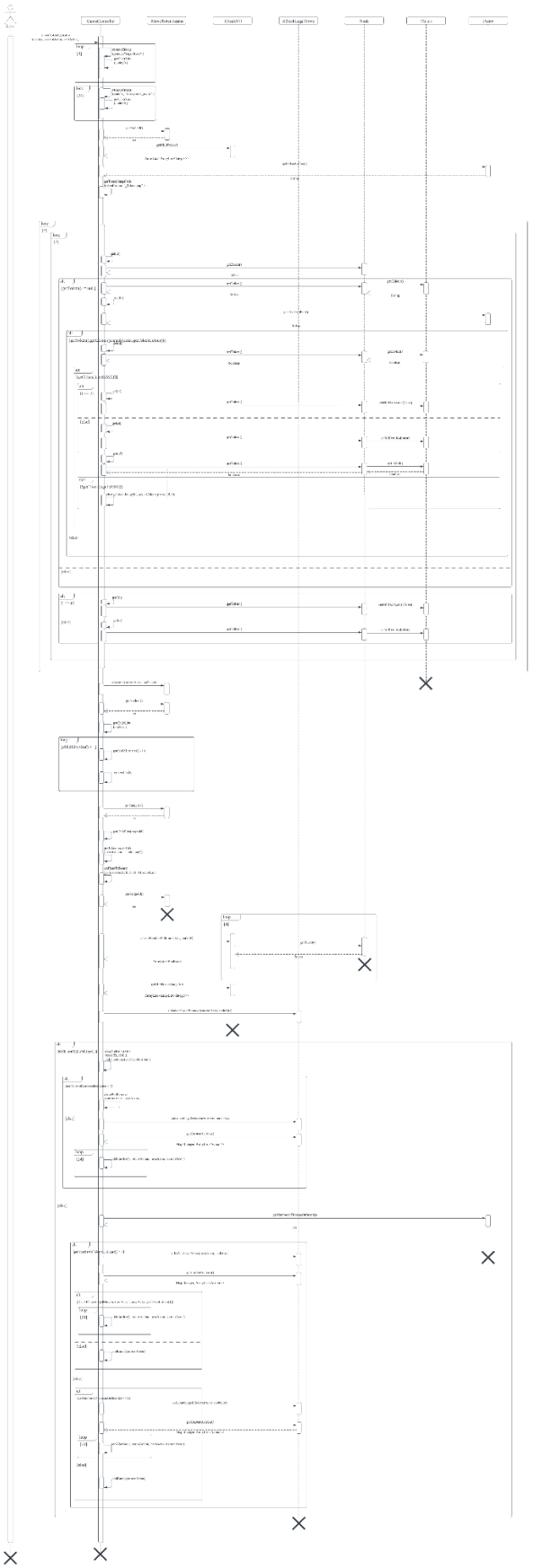
Human Value

One of the main human values that is highly relevant to the development of the Nine Men Morris gameplay and needs to be adhered to is the Equality human value. By having both players play under the same set of ruleset, fairness of gameplay can be assured at all times since there will not be any form of unfair advantage to any specific player. This is proven through how the legal moves are presented to the players as both players are only allowed to put 9 of their own tokens each on the board during the “Put Token” phase, and even if any player were to lose a token due to the formation of a mill by the other player, no additional tokens will be provided to the user who had lost a token during this phase. Both players have their legal moves during the “Move” phase calculated following the same conditionals and are only able to move a token to an empty adjacent node if they currently have more than three tokens on board. Both players have the equal opportunity to access the “Fly” token action when they have exactly three tokens on board, and are now able to move any of their own tokens to any empty nodes on the board. Both players are also allowed to only be able to remove an existing opponent token on board that is not part of a mill when they have formed a mill. As such, both players are provided equal opportunity of obtaining immunity to having their set of tokens that currently form a mill from being removed from the board. Overall both players are fully capable of conducting the same actions as the other player and are only ever provided a set of moves that strictly follows the predetermined set of gamerules. This assures that the game winner will always be determined in a fair manner where a winner can only be decided by forcing the opponent to either have no remaining legal moves or making their token count on board to be less than three.

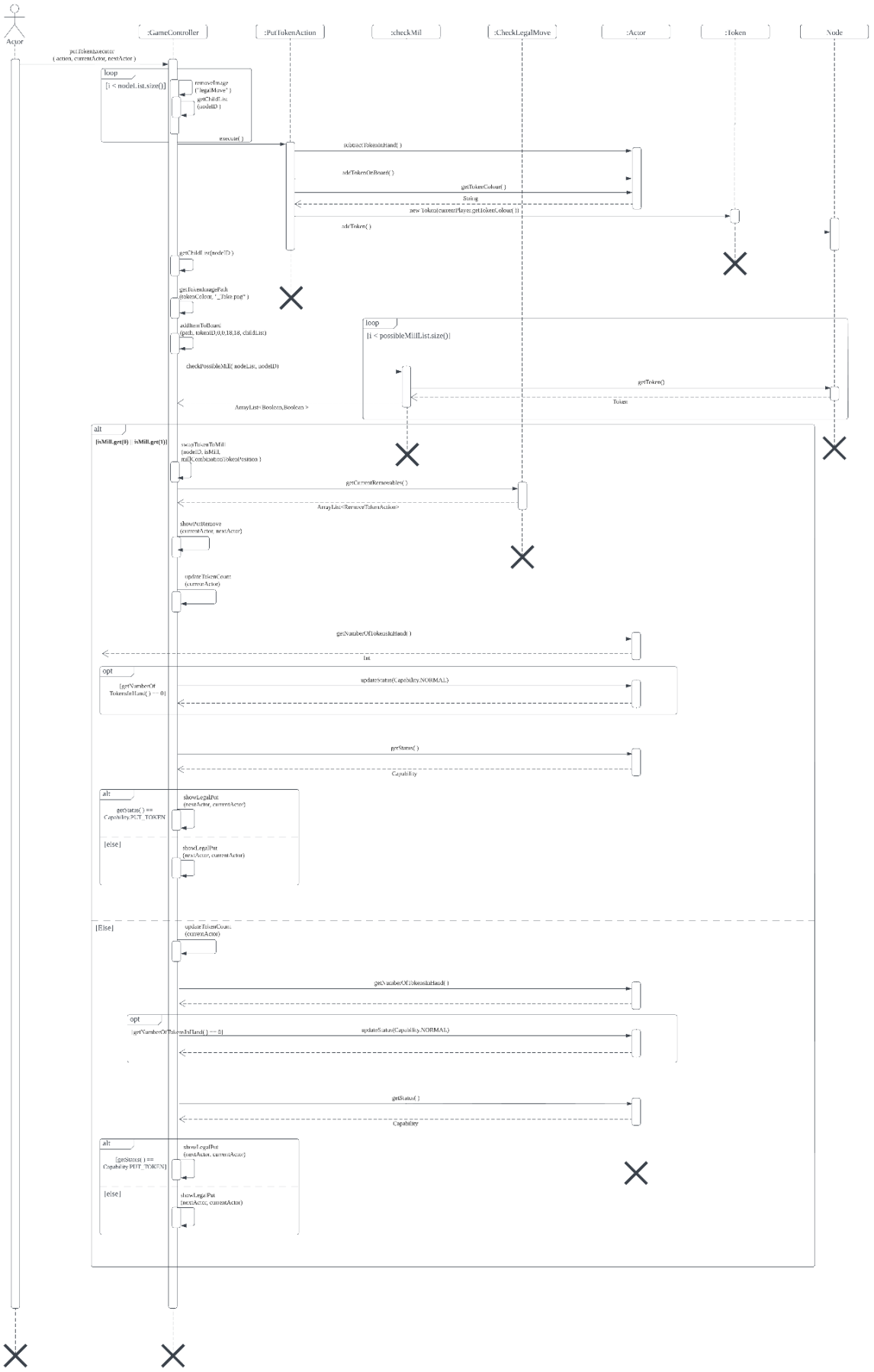
To further extend this human value, no player will be able to choose to go first but only limited to the selection of a side of the coin, either heads or tails. Once the players have selected a side of the coin, a coin would then be flipped and depending on the outcome, the player who goes first will be determined randomly at all times, ensuring that nobody gets any form of unfair advantage by being able to constantly select whether to go first or second when starting the game.

Sequence Diagram

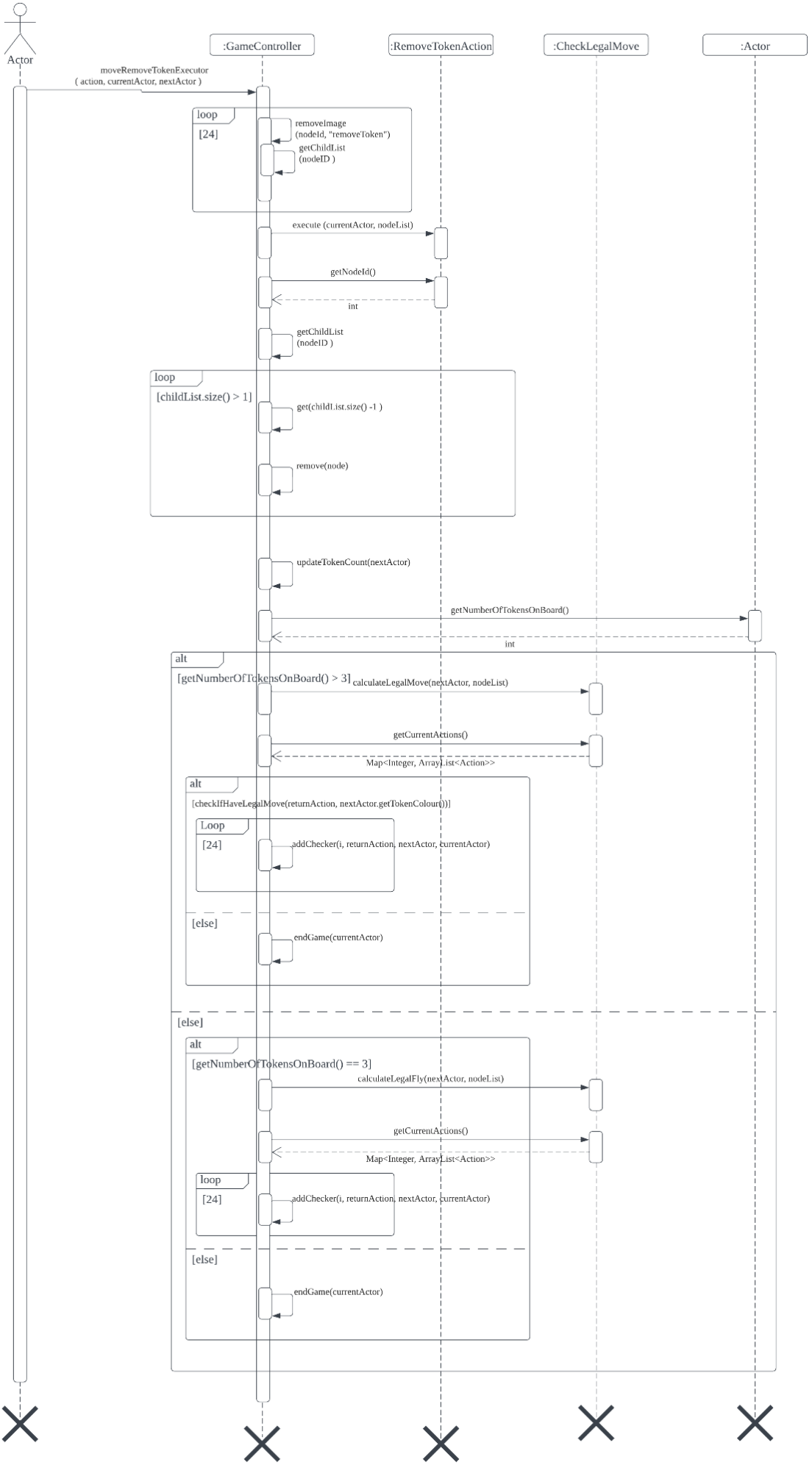
moveTokenExecutor sequence diagram



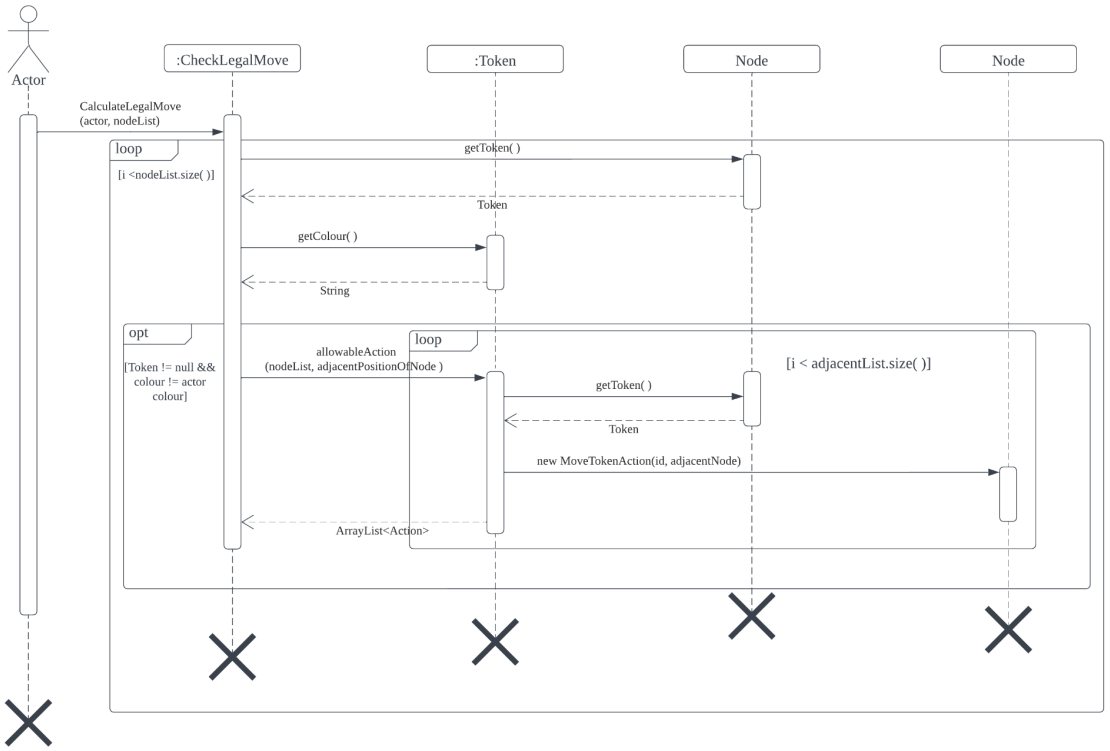
putTokenExecutor sequence diagram


Actor
:GameController
:PutTokenAction
:checkMil
:CheckLegalMove
:Actor
:Token
Node
putTokenExecute(action, currentActor, nextActor)
loop
[i < nodeList.size()]
removeImage ("legalMove")
getChildList(nodeID)
execute( )
subtractTokenInHand( )
addTokenToBoard( )
getTokenColour( )
String
new Token(currentPlayer.getTokenColour( ))
addToken( )
getChildList(nodeID)
getTokenImagePath (tokenColour, "_Tile.png")
addItemToBoard (path, tokenID,0,0,18,18, childList)
checkPossibleMill(nodeList, nodeID)
loop
[i < possibleMillList.size()]
getToken()
Token
ArrayList<Boolean,Boolean >
alt
[isMill.get(0) || isMill.get(1)]
sweepTokenToMill (nodeID, isMill, mill, combinationToken[version])
getCurrentRemoveables( )
ArrayList<RemoveTokenAction>
showPutRemove (currentActor, nextActor)
updateTokenCount (currentActor)
getNumberOfTokensInHand( )
Int
opt
[getNumberOf TokensInHand( ) == 0]
updateStatus(Capability.NORMAL)
getStatus( )
Capability
alt
getStatus( ) == Capability.PUT_TOKEN
showLegalPut (nextActor, currentActor)
[else]
showLegalPut (nextActor, currentActor)
[Else]
updateTokenCount (currentActor)
getNumberOfTokensInHand( )
opt
[getNumberOfTokensInHand( ) == 0]
updateStatus(Capability.NORMAL)
getStatus( )
Capability
alt
[getStatus( ) == Capability.PUT_TOKEN]
showLegalPut (nextActor, currentActor)
[else]
showLegalPut (nextActor, currentActor)

removeTokenExecutor sequence diagram


Actor
:GameController
:RemoveTokenAction
:CheckLegalMove
:Actor
moveRemoveTokenExecutor
(action, currentActor, nextActor)
loop
[24]
removeImage
(nodeId, "removeToken")
getChildList
(nodeID)
execute (currentActor, nodeList)
getNodeId()
int
getChildList
(nodeID)
loop
[childList.size() > 1]
get(childList.size() -1)
remove(node)
updateTokenCount(nextActor)
getNumberOfTokensOnBoard()
int
alt
[getNumberOfTokensOnBoard() > 3]
calculateLegalMove(nextActor, nodeList)
getCurrentActions()
Map<Integer, ArrayList<Action>>
alt
[checkIfHaveLegalMove(returnAction, nextActor.getTokenColour())]
Loop
[24]
addChecker(i, returnAction, nextActor, currentActor)
[else]
endGame(currentActor)
[else]
alt
[getNumberOfTokensOnBoard() == 3]
calculateLegalFly(nextActor, nodeList)
getCurrentActions()
Map<Integer, ArrayList<Action>>
loop
[24]
addChecker(i, returnAction, nextActor, currentActor)
[else]
endGame(currentActor)

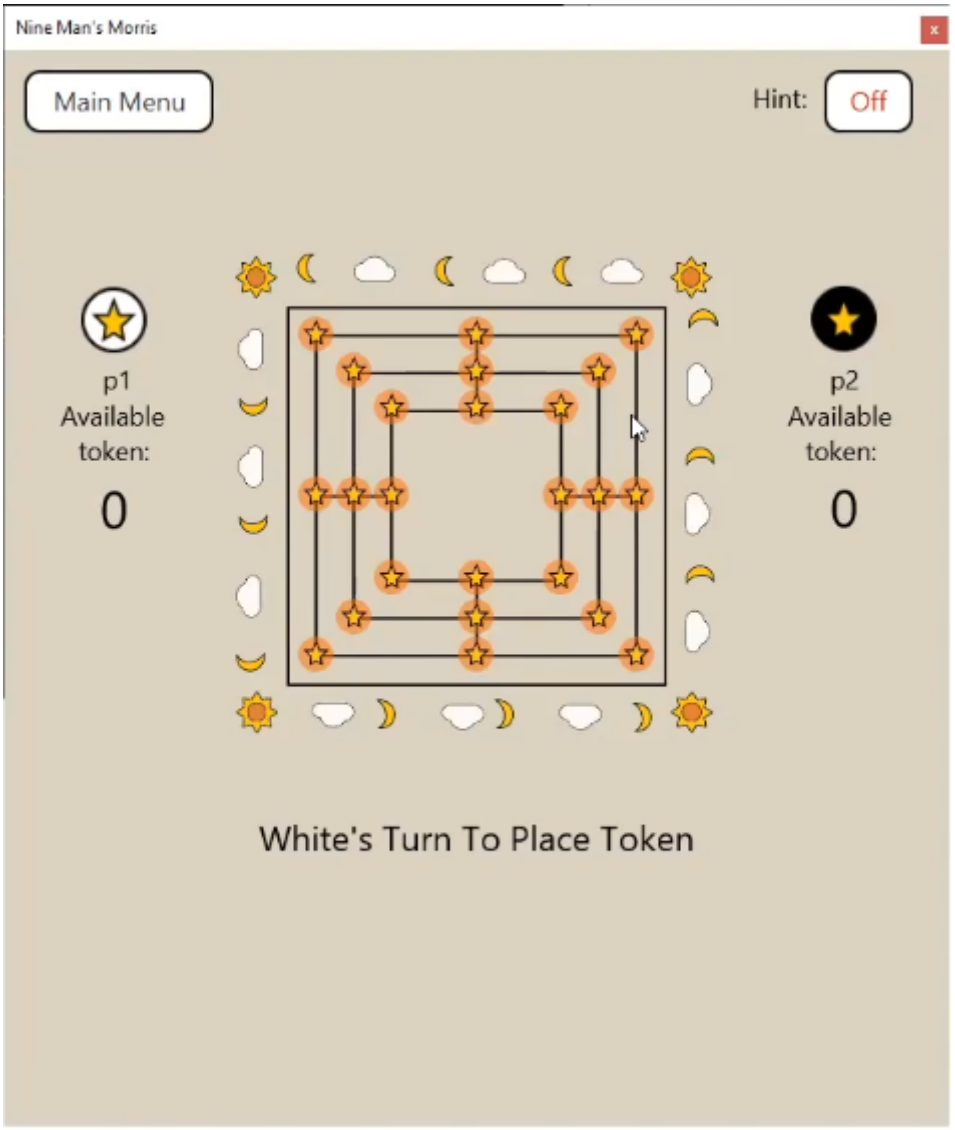
checkLegalMove sequence diagram



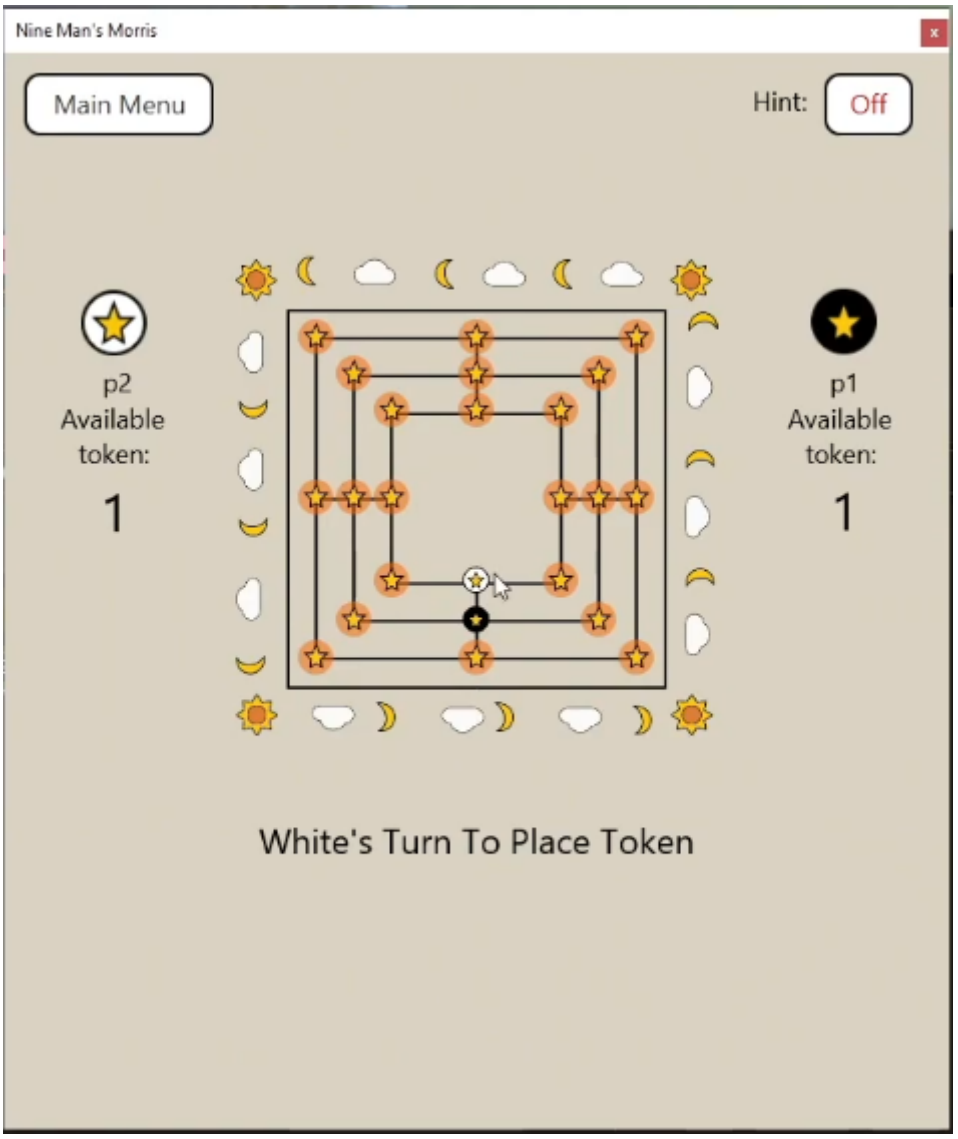
Link to HD version of the sequence diagrams:
https://drive.google.com/drive/folders/1sD5OmvjYZx4I9THuMaqCWDAd9sNY1QM0?usp=share_link
https://lucid.app/lucidchart/a0683381-bc6d-4cff-90f6-c9f1d0d63ccb/edit?view_items=IYtXvTUQDTSf&invitationId=inv_b5efde6b-51cc-41eb-ac29-50807bebf07a

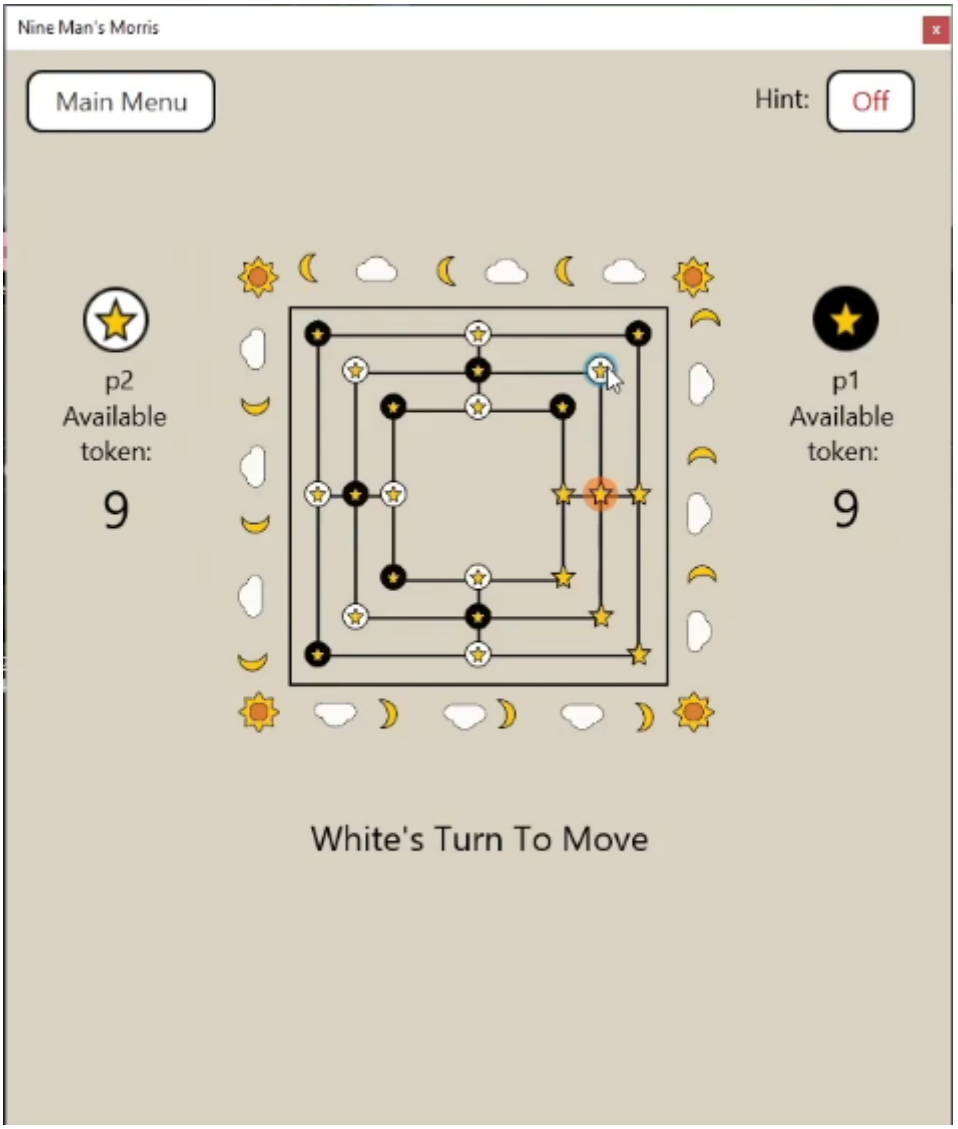
Screenshots of the five stages

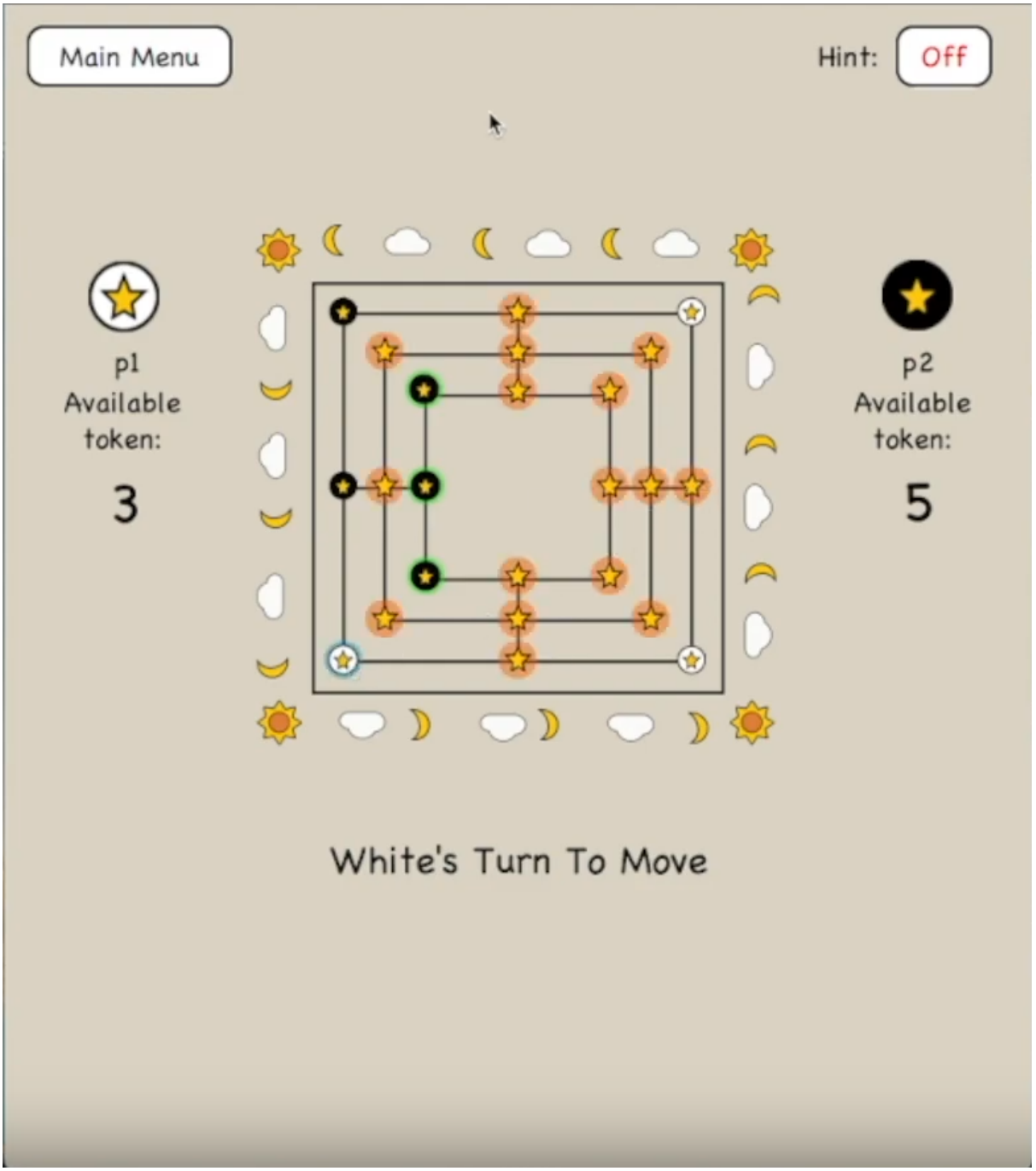
Empty board



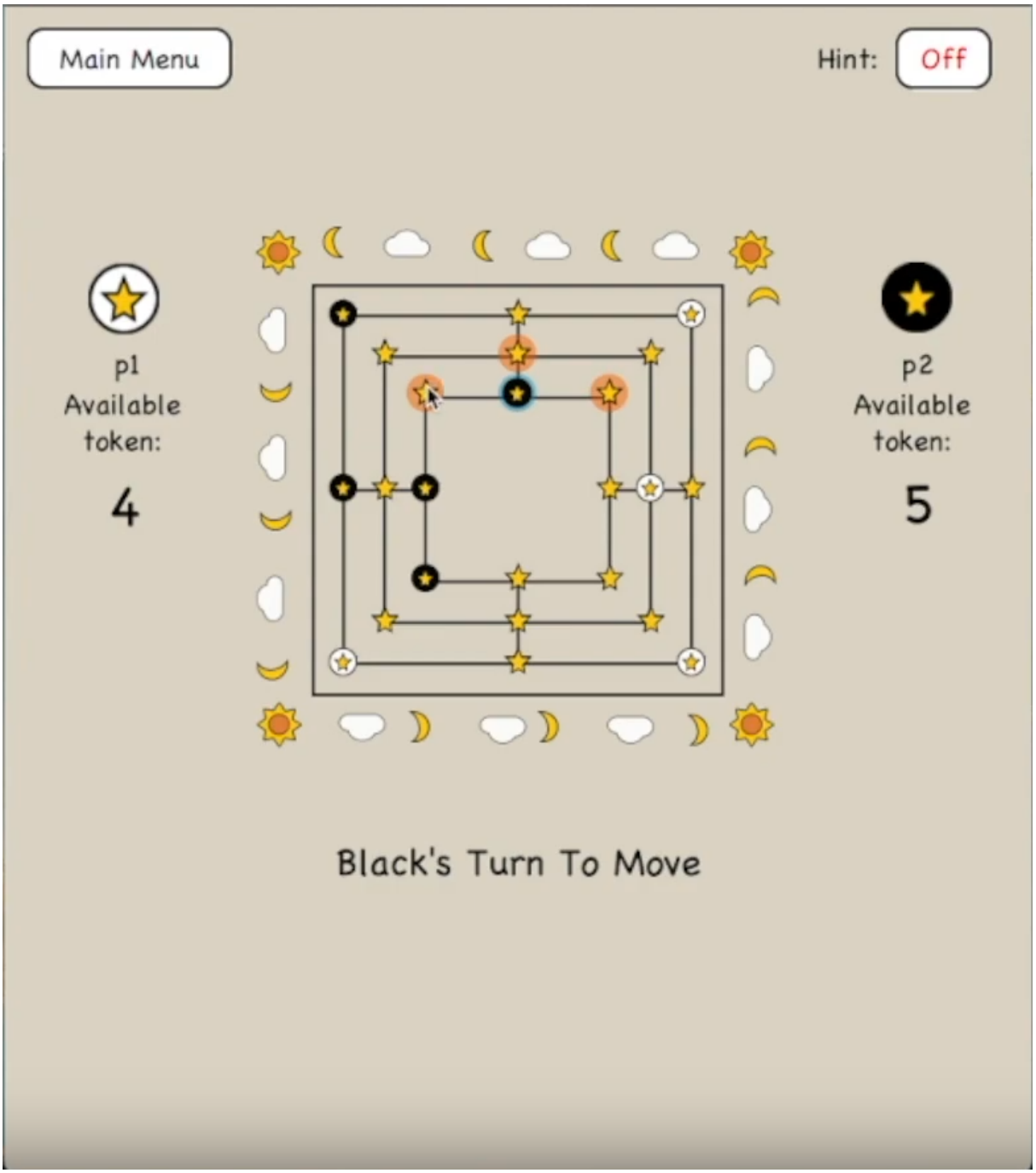
Put, Move, Fly stages

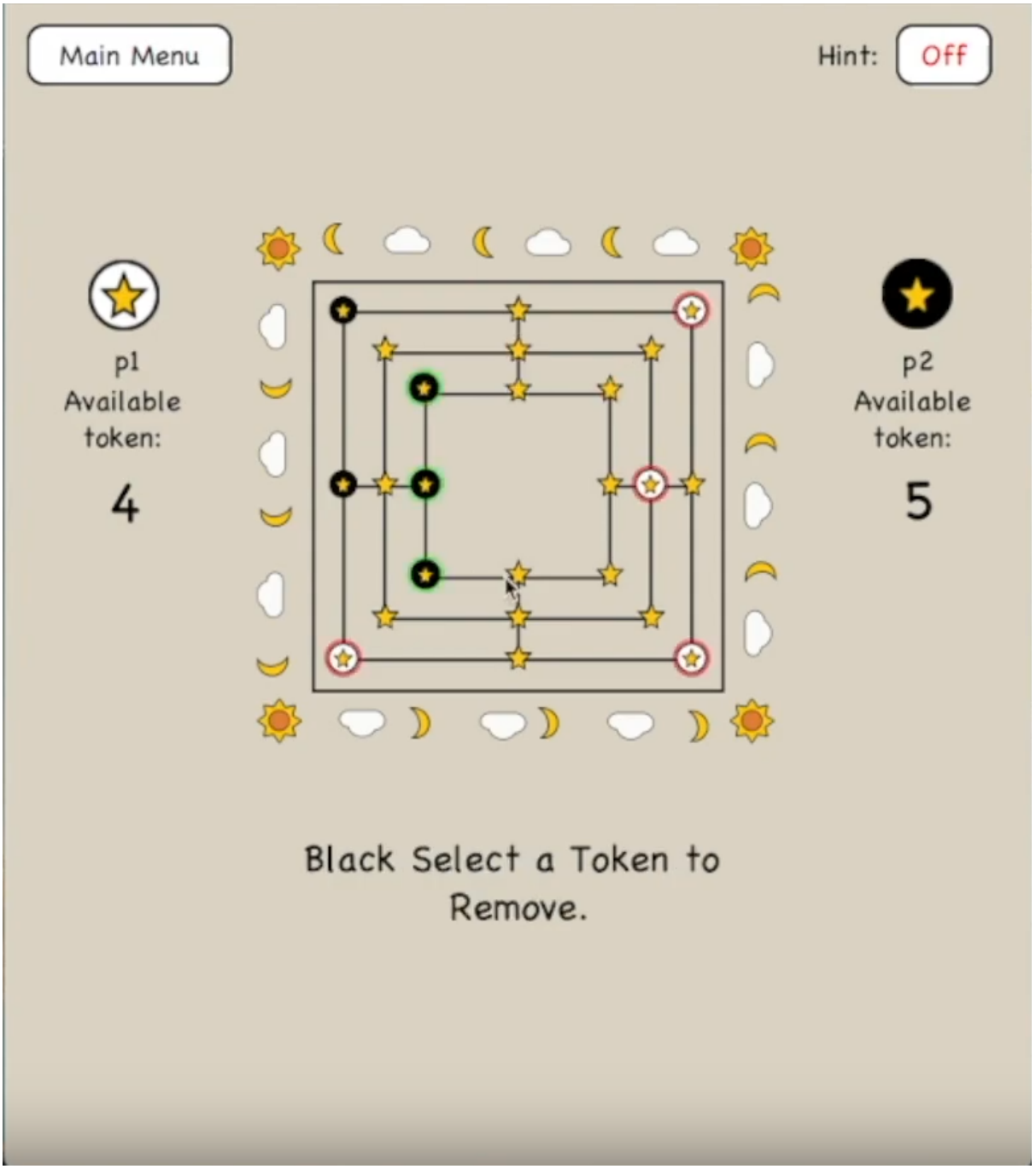



Nine Man's Morris
Main Menu
Hint:
Off
p2
Available
token:
9
p1
Available
token:
9
White's Turn To Move

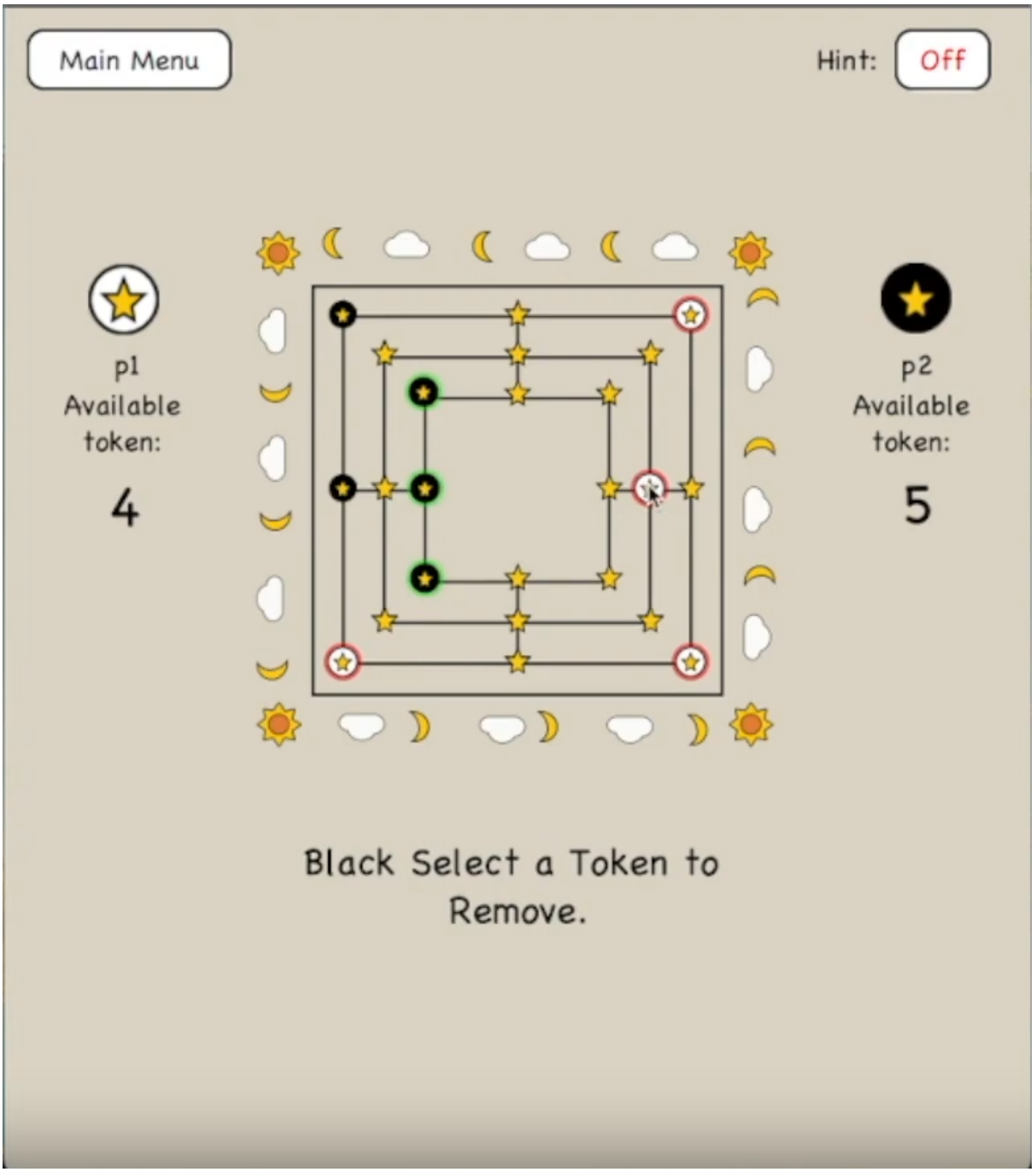

Main Menu
Hint:
Off
p1
Available
token:
3
p2
Available
token:
5
White's Turn To Move

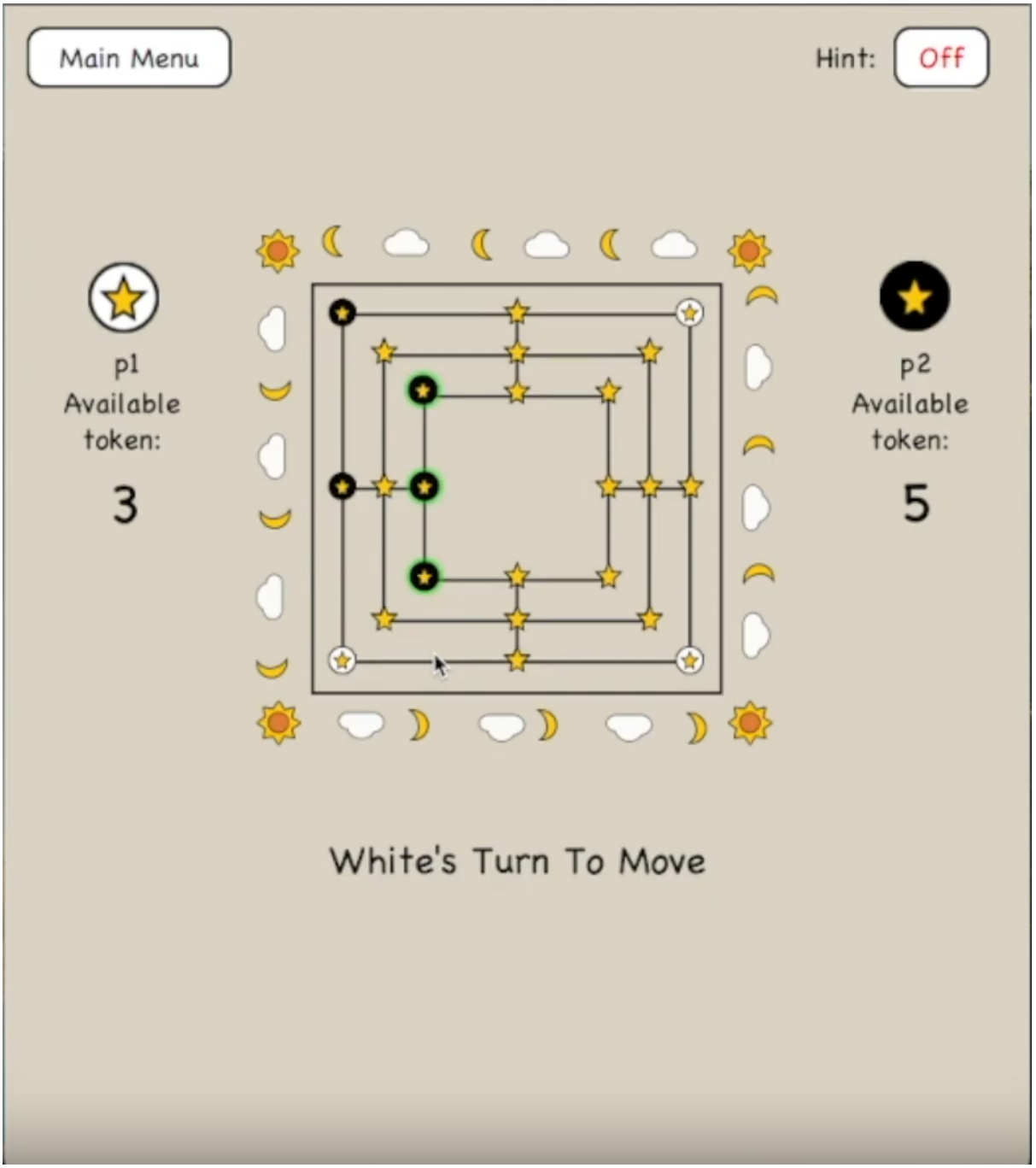
Detection of new mill



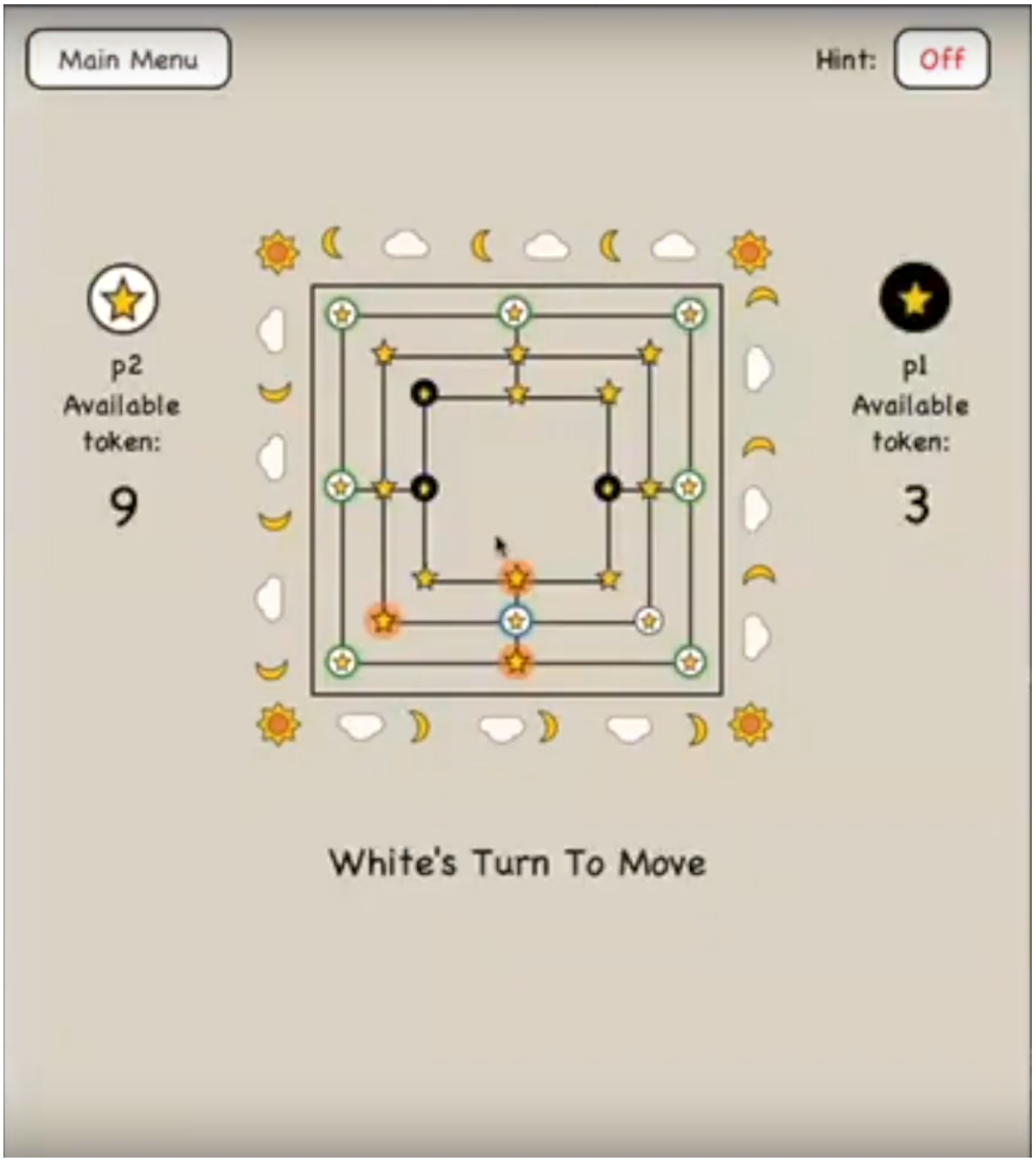

Main Menu
Hint:
Off
p1
Available
token:
4
p2
Available
token:
5
Black Select a Token to
Remove.

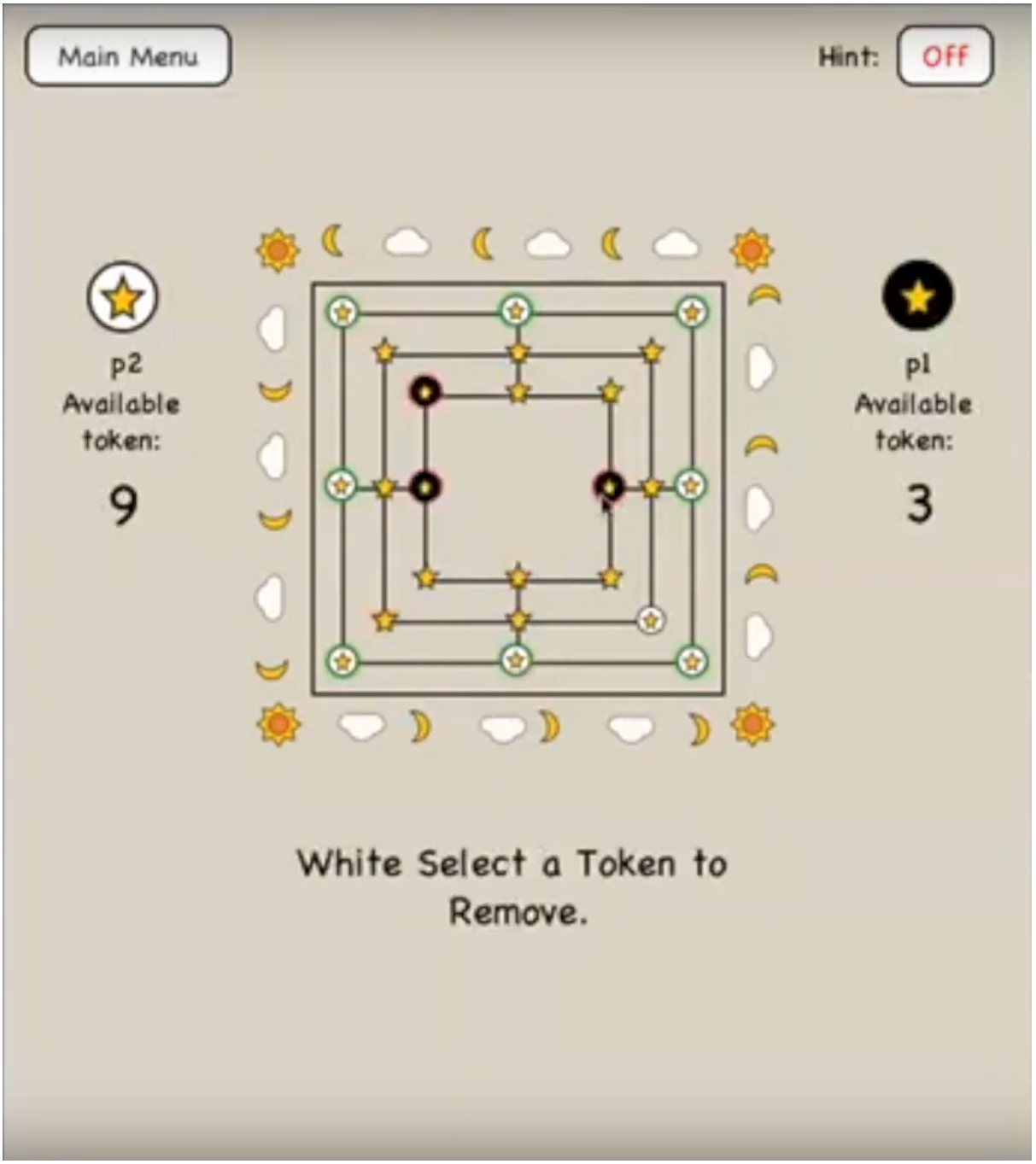
Correct removal of token after forming a mill

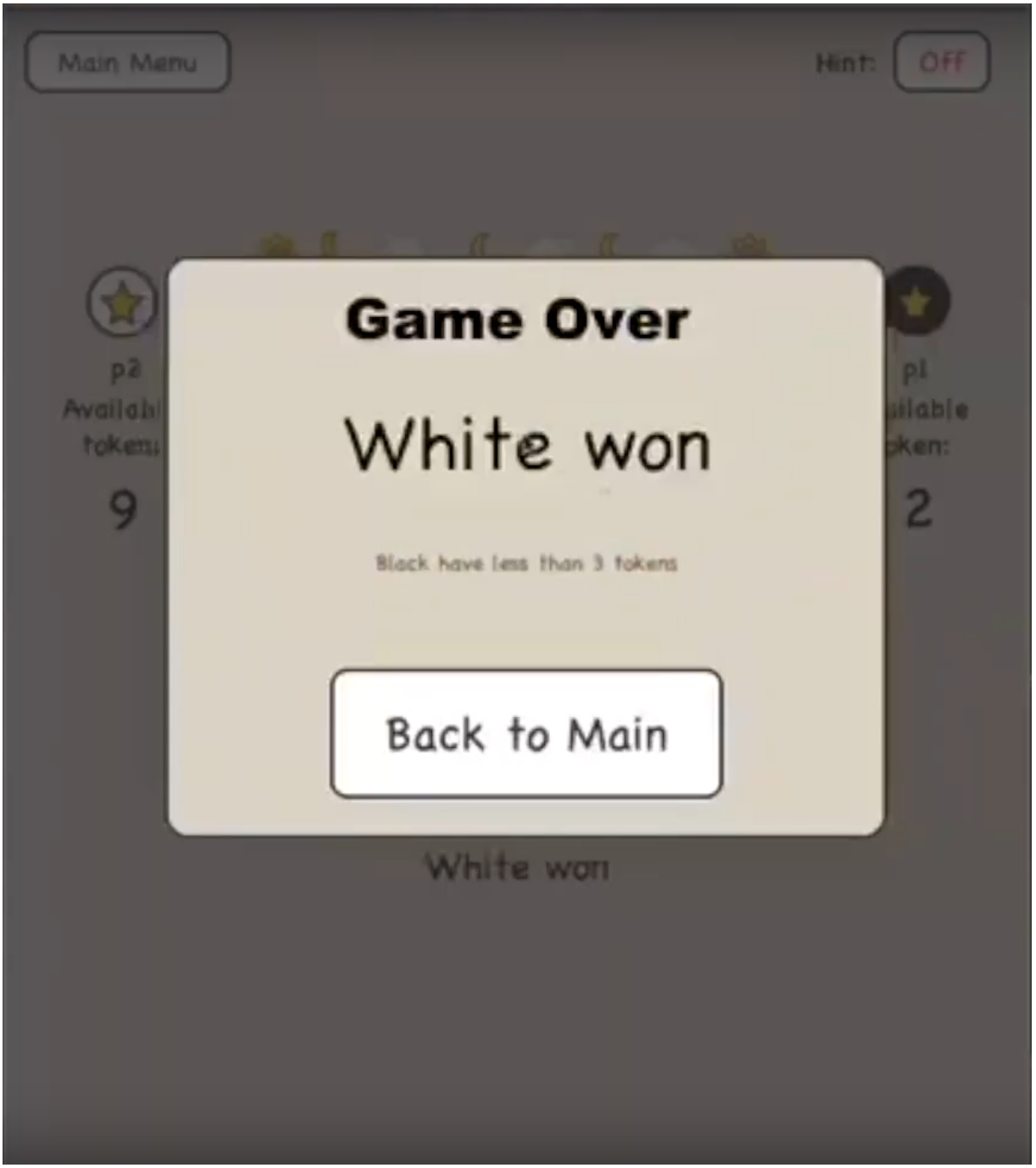


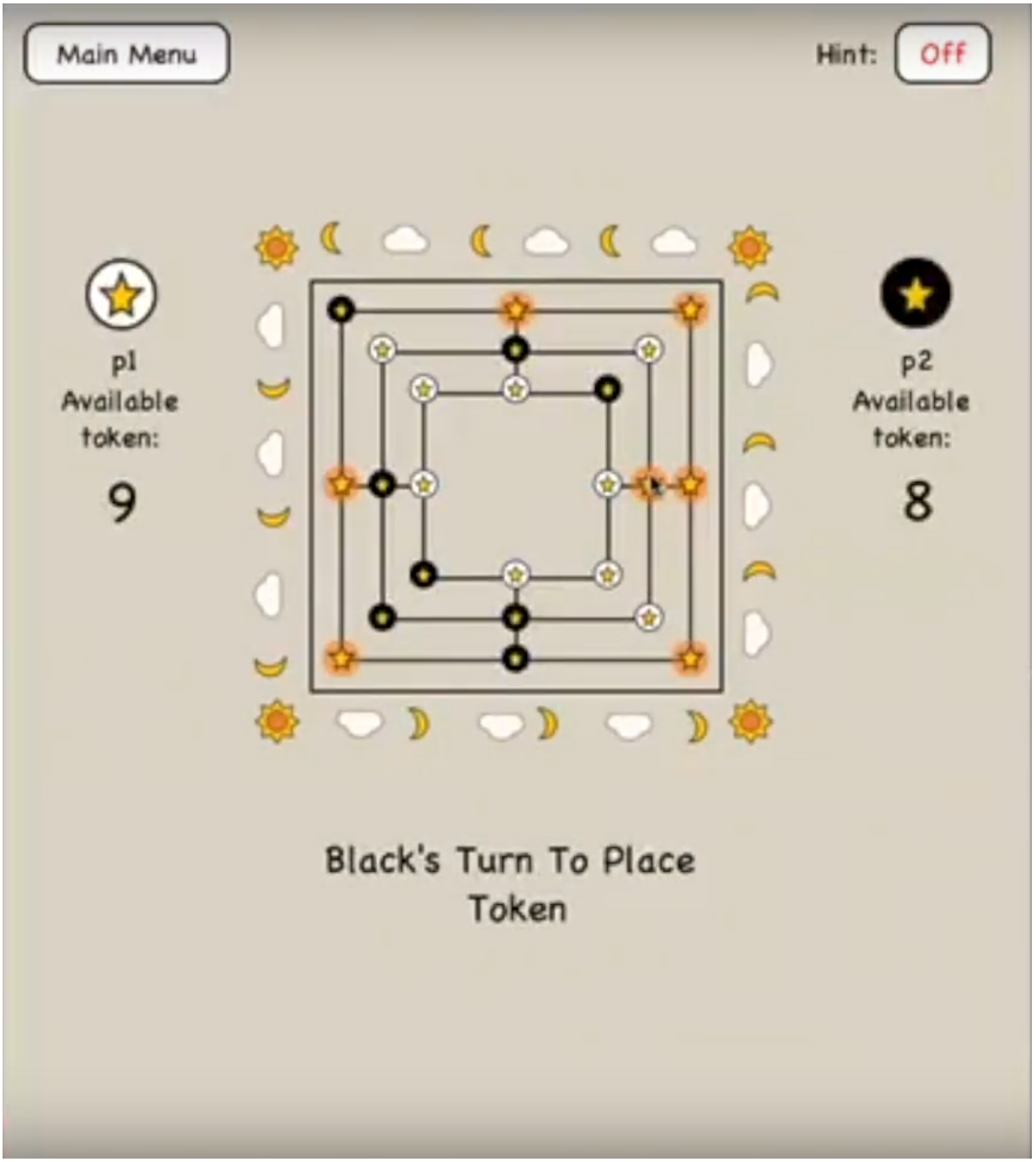

Main Menu
Hint:
Off
p1
Available
token:
3
p2
Available
token:
5
White's Turn To Move

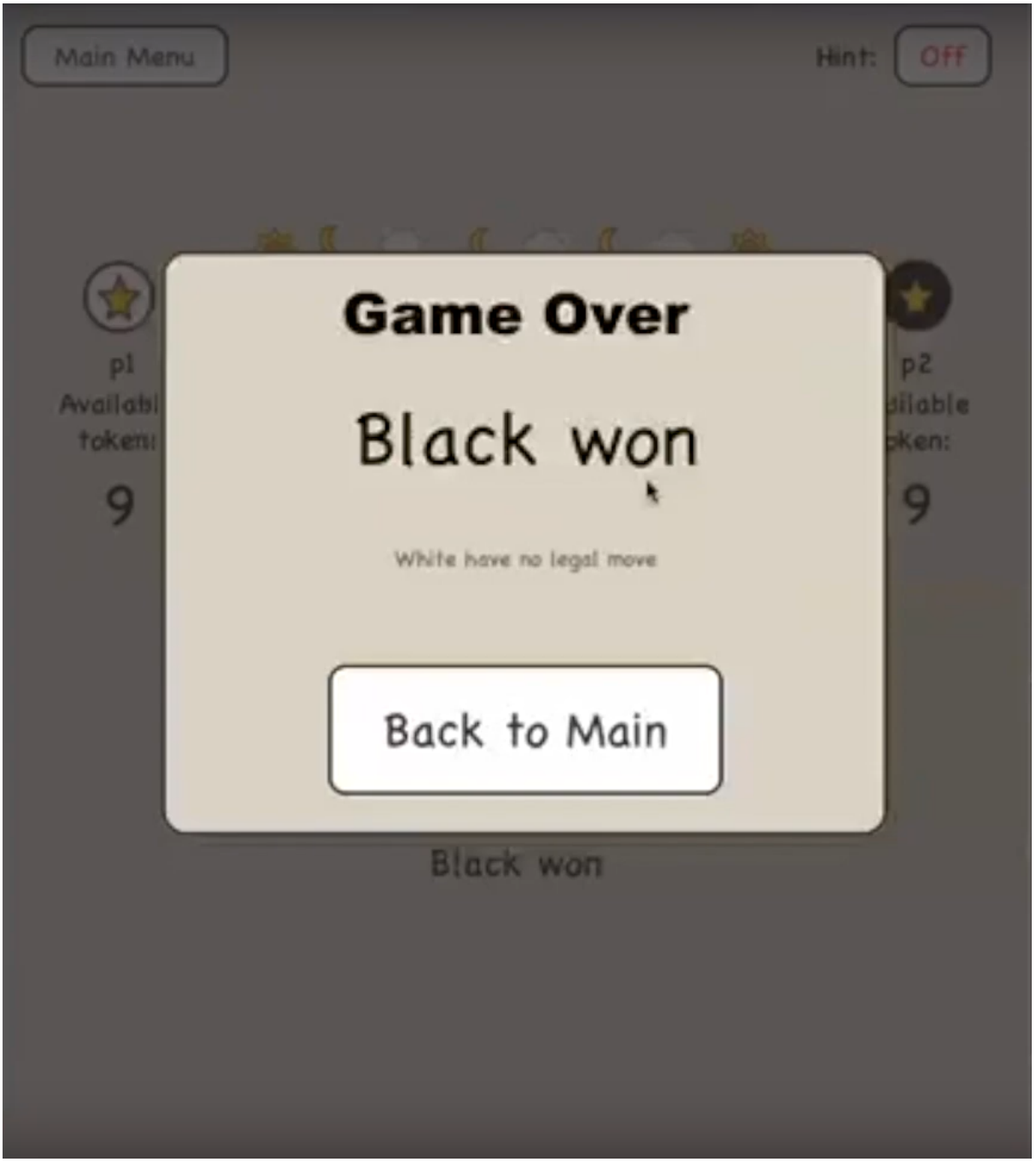
Detection of endgame




Main Menu
Hint:
OFF
p2
Available
token:
9
p1
Available
token:
3
White Select a Token to
Remove.


Main Menu
Hint:
OFF
Game Over
White won
Black have less than 3 tokens
Back to Main
9
2
White won


Main Menu
Hint:
Off
p1
Available
token:
9
p2
Available
token:
8
Black's Turn To Place
Token


Main Menu
Hint:
OFF
Game Over
Black won
White have no legal move
Back to Main
p1
9
p2
9
Black won