FIT 3077

Semester 1

Architecture and Design Relation

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

Table of Contents

Design Rational

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

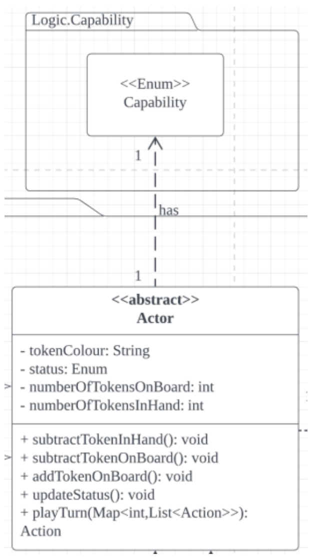
The Node class has an association with the Token class, as in each cycle, the Board class must examine each node in the list to determine all legal moves that a player may make. To determine whether any legal moves exist on a given node, the node must check if any tokens are present on it. Therefore, 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

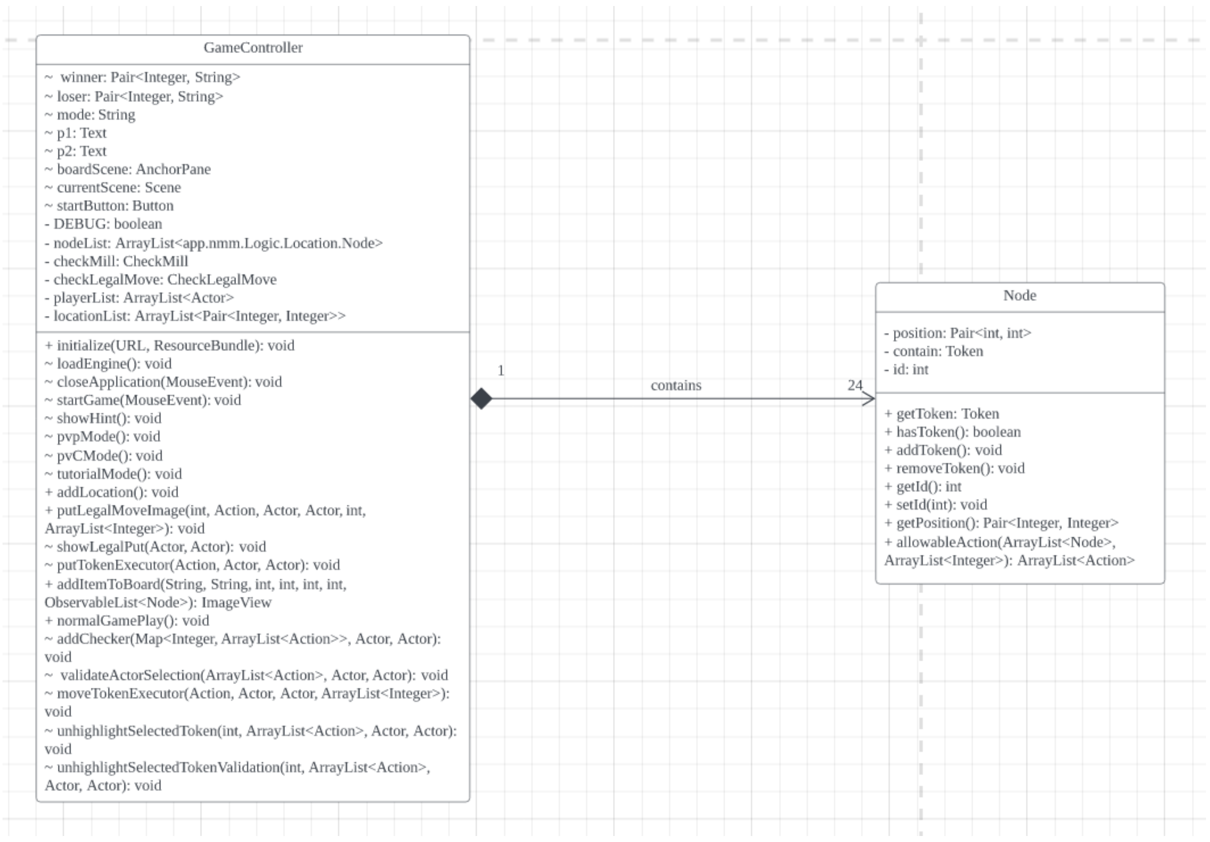

Logic.Capability
<<Enum>>
Capability
1
has
1
<<abstract>>
Actor
- tokenColour: String
- status: Enum
- numberOfTokensOnBoard: int
- numberOfTokensInHand: int
+ subtractTokenInHand(): void
+ subtractTokenOnBoard(): void
+ addTokenOnBoard(): void
+ updateStatus(): void
+ playTurn(Map<int,List<Action>>): Action

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.

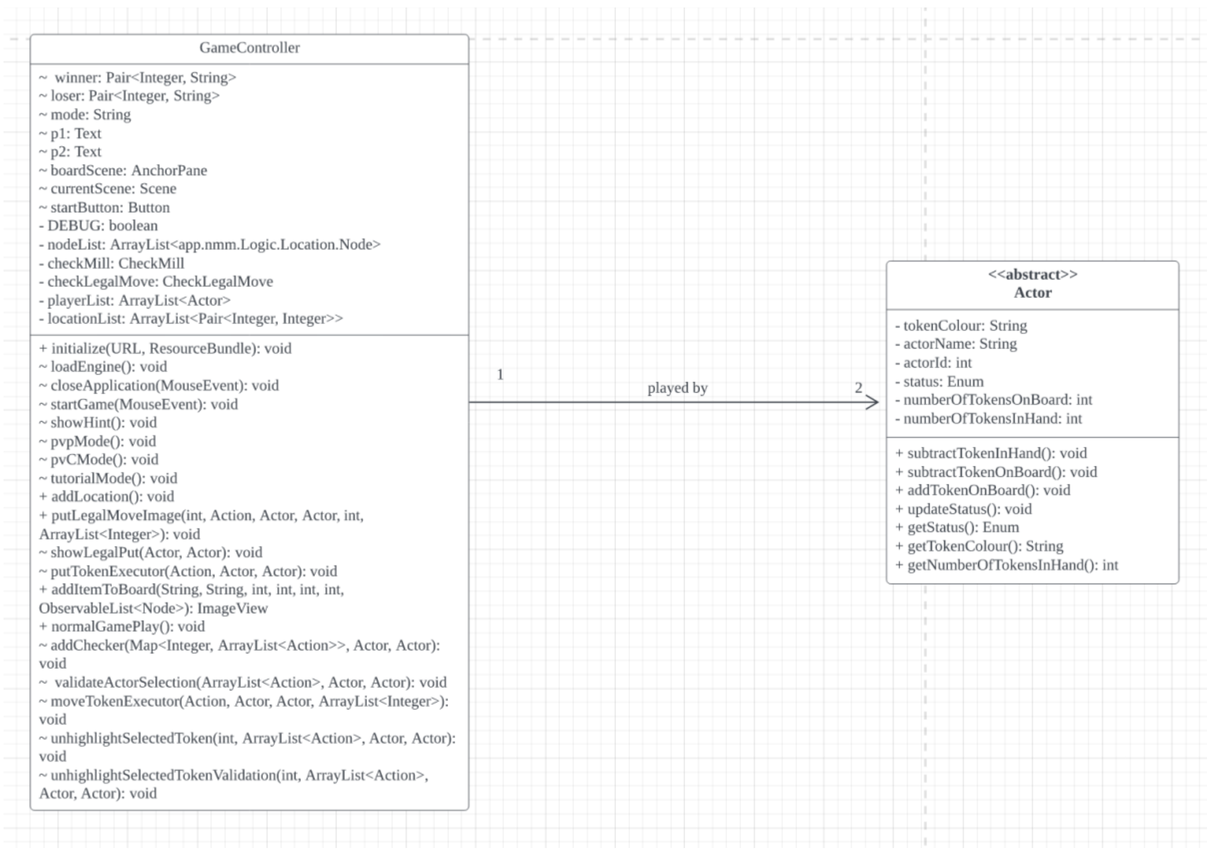
GameController and Node


GameController
~ winner: Pair<Integer, String>
~ loser: Pair<Integer, String>
~ mode: String
~ p1: Text
~ p2: Text
~ boardScene: AnchorPane
~ currentScene: Scene
~ startButton: Button
- DEBUG: boolean
- nodeList: ArrayList<app.nmm.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
~ pvpMode(): 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
~ addChecker(Map<Integer, ArrayList<Action>>, Actor, Actor): void
~ validateActorSelection(ArrayList<Action>, Actor, Actor): void
~ moveTokenExecutor(Action, Actor, Actor, ArrayList<Integer>): void
~ unhighlightSelectedToken(int, ArrayList<Action>, Actor, Actor): void
~ unhighlightSelectedTokenValidation(int, ArrayList<Action>, Actor, Actor): void
1
contains
24
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>

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.

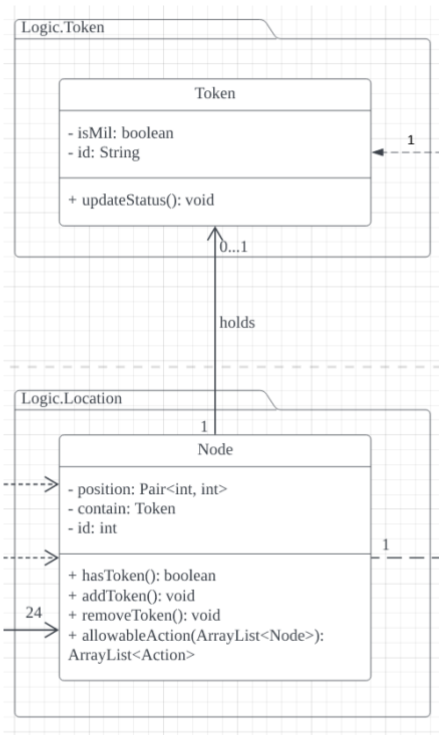
In a Nine Man Morris game, the board consists of 24 intersections, which are nodes connected by straight lines, with each Node connecting to 1- 4 neighbouring nodes depending on its position. Therefore, the GameController class will have 24 instances of the Node class in the constructor, represented 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
~ winner: Pair<Integer, String>
~ loser: Pair<Integer, String>
~ mode: String
~ p1: Text
~ p2: Text
~ boardScene: AnchorPane
~ currentScene: Scene
~ startButton: Button
- DEBUG: boolean
- nodeList: ArrayList<app.nmm.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
~ pvpMode(): 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
~ addChecker(Map<Integer, ArrayList<Action>>, Actor, Actor): void
~ validateActorSelection(ArrayList<Action>, Actor, Actor): void
~ moveTokenExecutor(Action, Actor, Actor, ArrayList<Integer>): void
~ unhighlightSelectedToken(int, ArrayList<Action>, Actor, Actor): void
~ unhighlightSelectedTokenValidation(int, ArrayList<Action>, Actor, Actor): void
1
played by
2
<<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

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



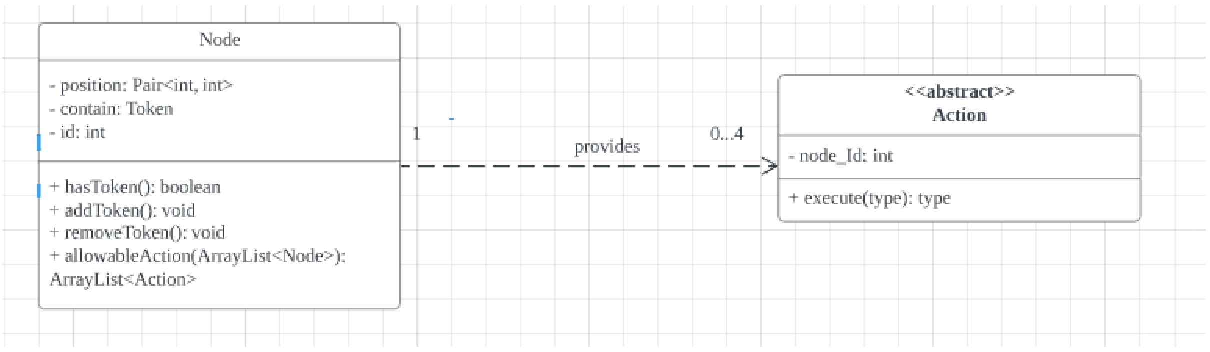
The multiplicity between Node class and Token class is 1 -> 0…1 because each Node on the Board may or may not have a Token placed on it.

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

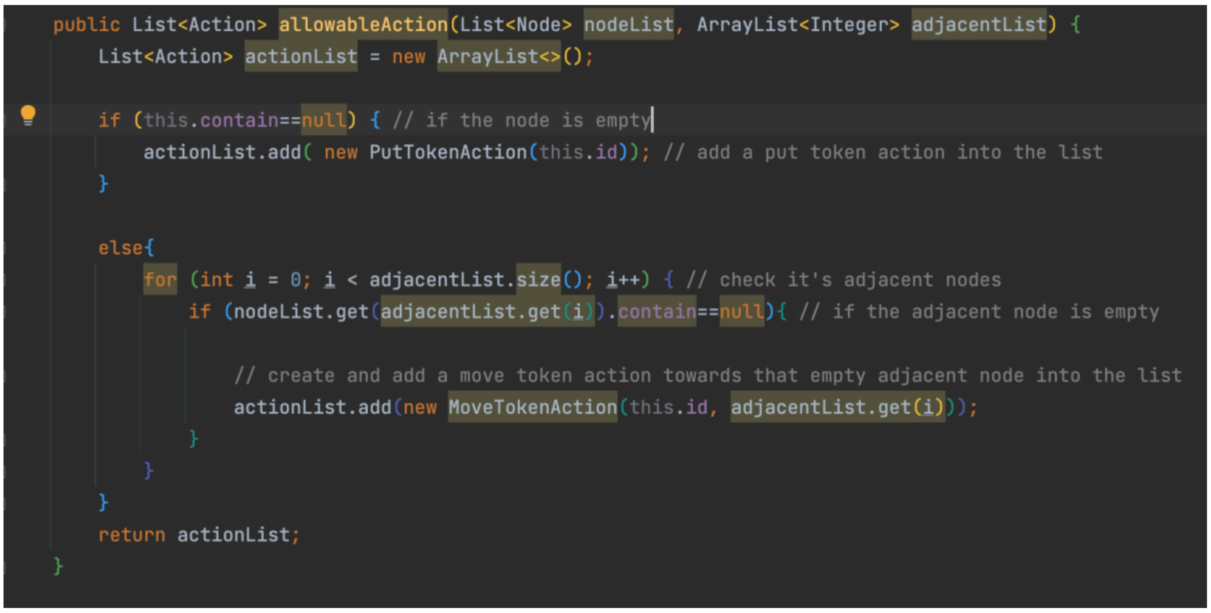


The multiplicity between Node class and Action class is 1 -> 0…4 because a Node can give an Actor up to 4 allowable actions to be performed. In other words, the allowableAction method returns 0 to 4 instances of Action.

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.

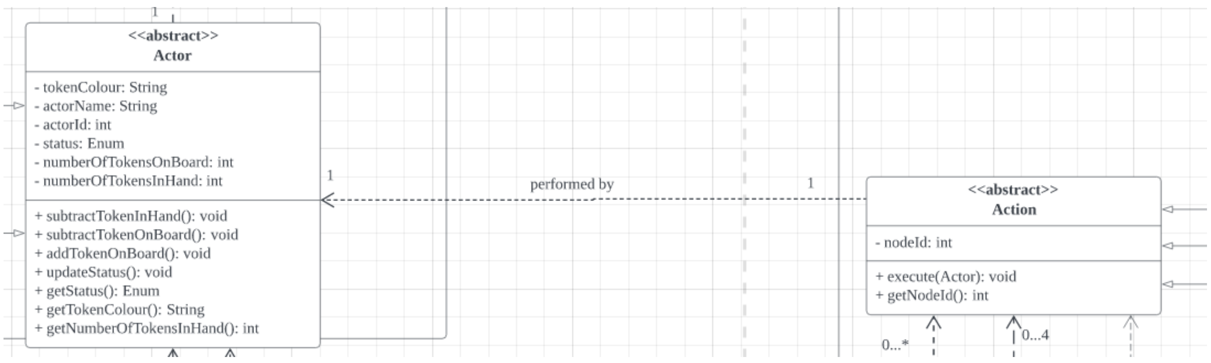


```java
public List<Action> allowableAction(List<Node> nodeList, ArrayList<Integer> adjacentList) {
    List<Action> actionList = new ArrayList<>();

    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.

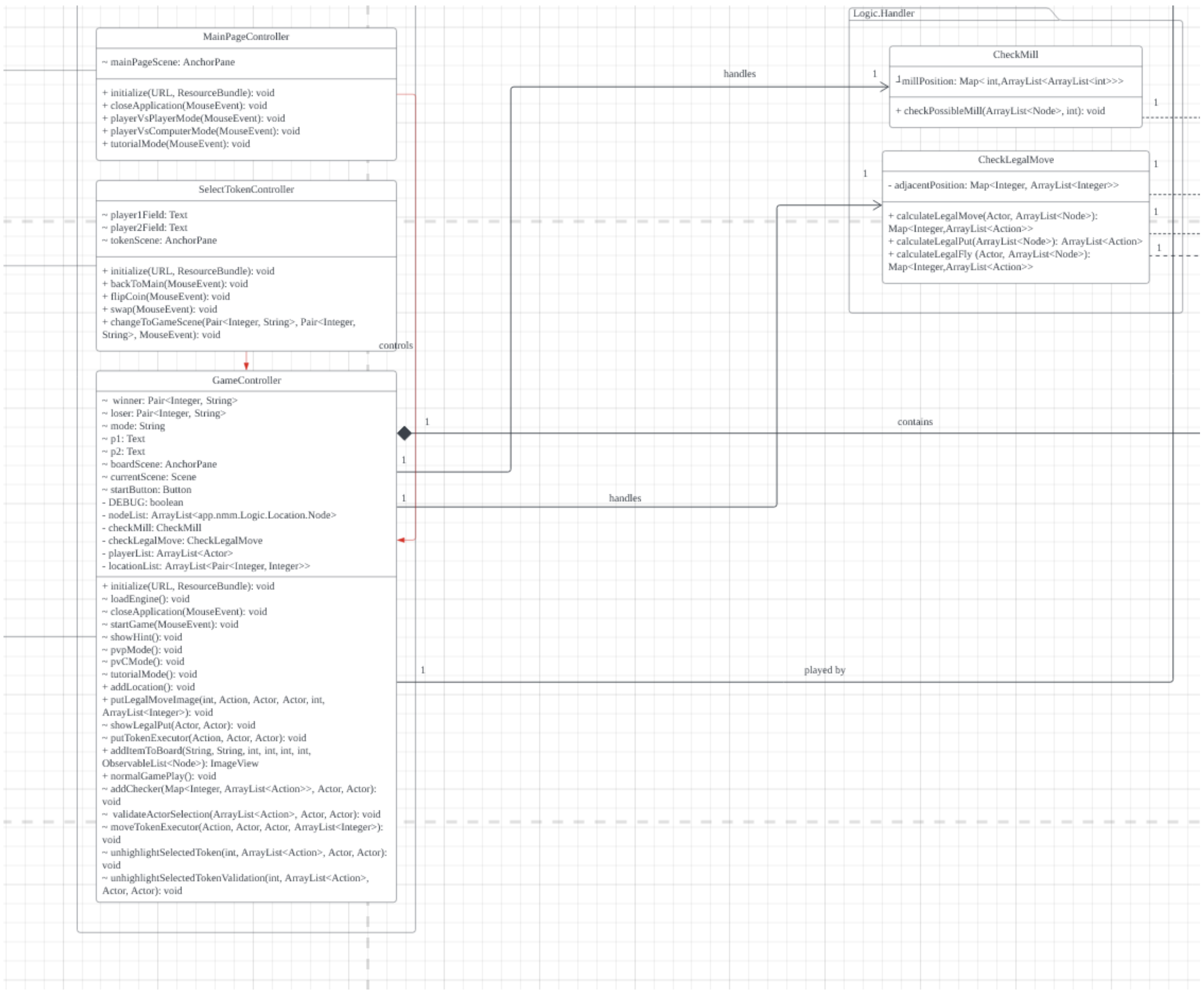
Hence the allowableAction method in the Node class can return a minimum of 0 Action and a maximum of 4 Action instances in the list.

Action and Actor



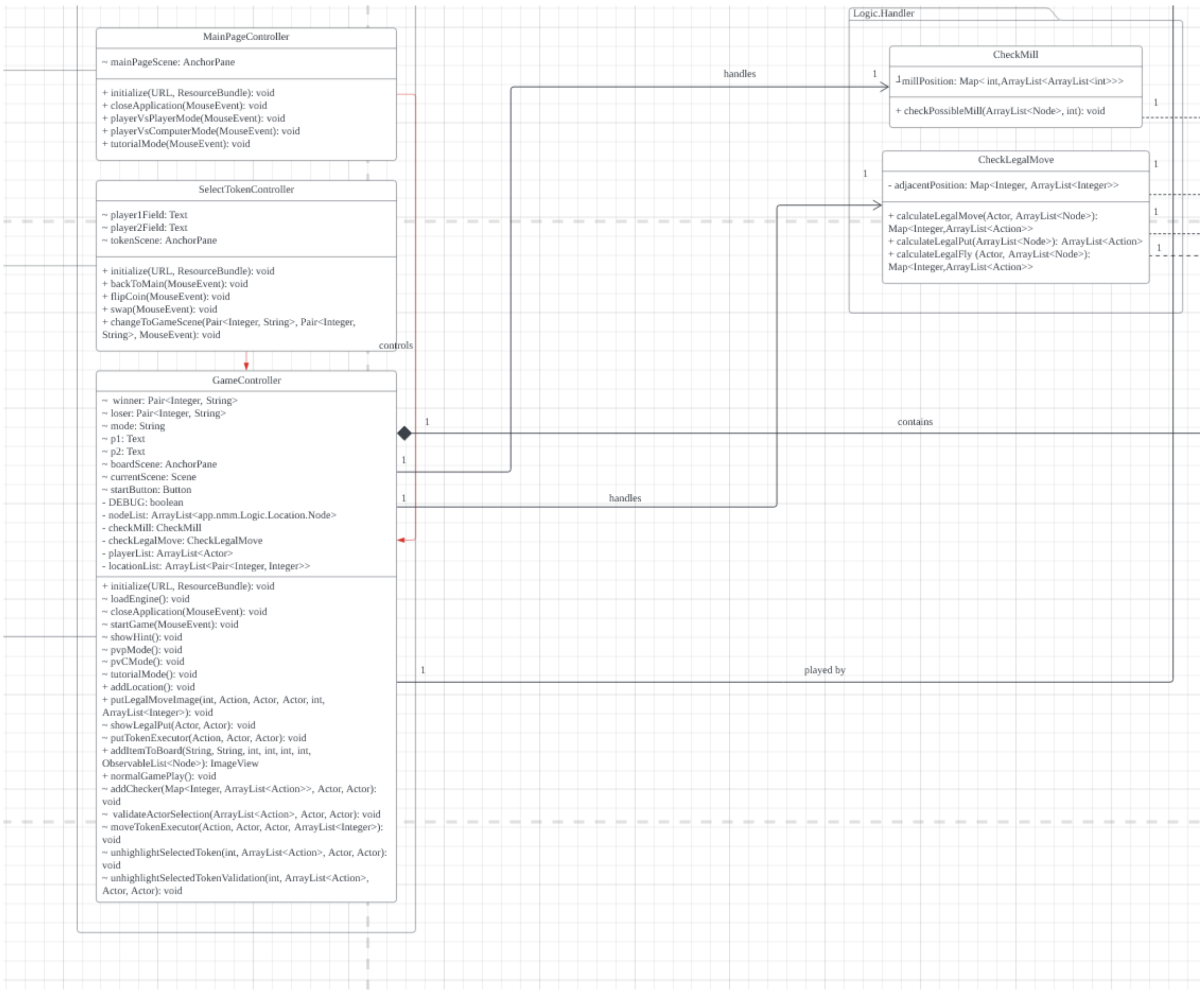
The multiplicity between Action and Actor is 1 -> 1 because the execute method in the Action class references one instance of Actor in the method parameter. In other words, an Action can only be performed by one Actor at a time. The execute method in the Action class changes the attributes in the Actor 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


Logic.Handler
CheckMill
1millPosition: Map< int,ArrayList<ArrayList<int>>>
+ checkPossibleMill(ArrayList<Node>, int): void
CheckLegalMove
- adjacentPosition: 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>>
MainPageController
~ mainPageScene: AnchorPane
+ initialize(URL, ResourceBundle): void
+ closeApplication(MouseEvent): void
+ playerVsPlayerMode(MouseEvent): void
+ playerVsComputerMode(MouseEvent): void
+ tutorialMode(MouseEvent): void
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
controls
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<app.nmm.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
~ pvpMode(): 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
~ addChecker(Map<Integer, ArrayList<Action>>, Actor, Actor): void
~ validateActorSelection(ArrayList<Action>, Actor, Actor): void
~ moveTokenExecutor(Action, Actor, Actor, ArrayList<Integer>): void
~ unhighlightSelectedToken(int, ArrayList<Action>, Actor, Actor): void
~ unhighlightSelectedTokenValidation(int, ArrayList<Action>, Actor, Actor): void
handles
handles
contains
played by
1
1
1
1
1
1
1
1
1
1
1

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.

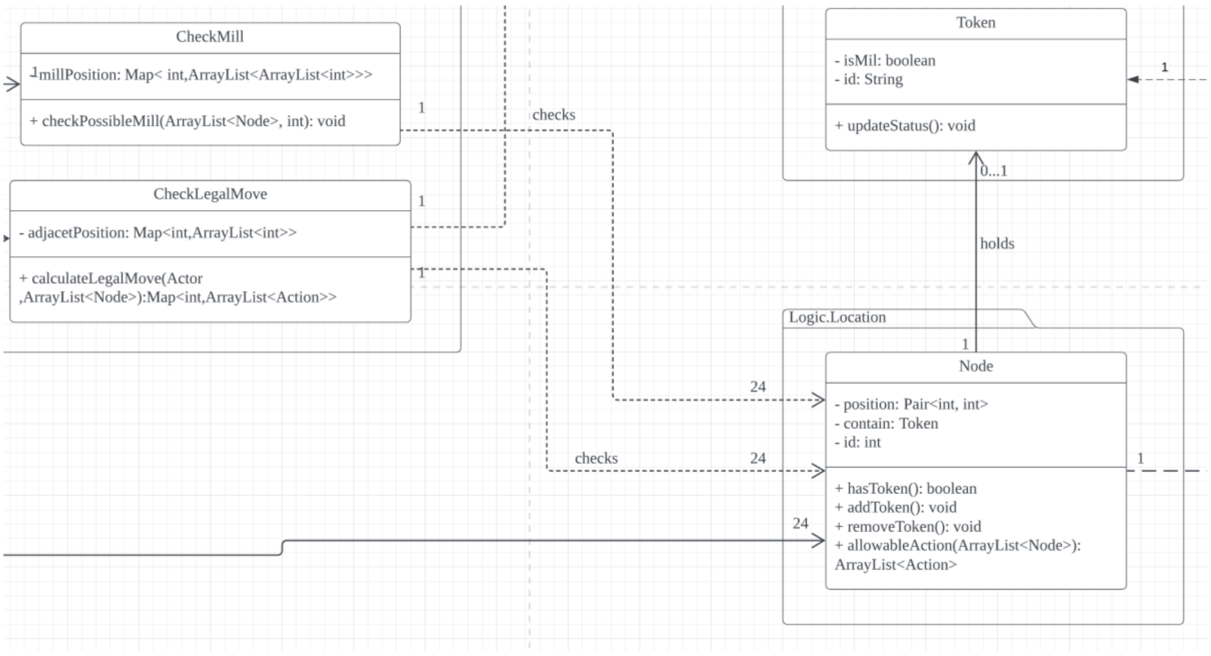
GameController to CheckLegalMove



MainPageController
~ mainPageScene: AnchorPane
+ initialize(URL, ResourceBundle): void
+ closeApplication(MouseEvent): void
+ playerVsPlayerMode(MouseEvent): void
+ playerVsComputerMode(MouseEvent): void
+ tutorialMode(MouseEvent): void
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
controls
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<app.nmm.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
~ pvpMode(): 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
~ addChecker(Map<Integer, ArrayList<Action>>, Actor, Actor): void
~ validateActorSelection(ArrayList<Action>, Actor, Actor): void
~ moveTokenExecutor(Action, Actor, Actor, ArrayList<Integer>): void
~ unhighlightSelectedToken(int, ArrayList<Action>, Actor, Actor): void
~ unhighlightSelectedTokenValidation(int, ArrayList<Action>, Actor, Actor): void
1
contains
1
handles
1
handles
1
played by
Logic.Handler
CheckMill
1
1millPosition: Map< int,ArrayList<ArrayList<int>>>
+ checkPossibleMill(ArrayList<Node>, int): void
1
CheckLegalMove
1
1
- adjacentPosition: 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>>
1
1

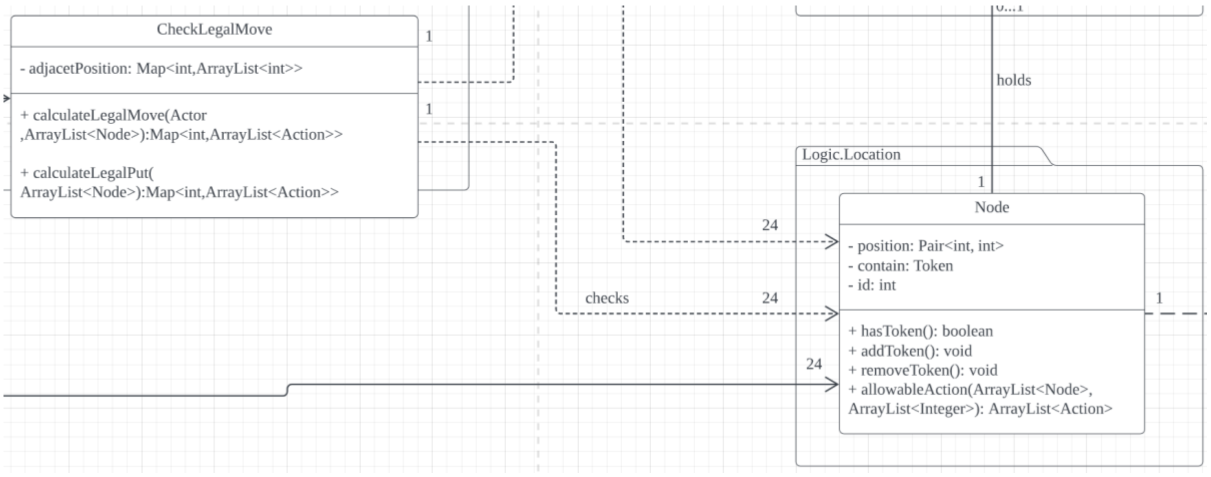
The multiplicity from GameController to CheckLegalMove class is 1 -> 1 because there will only be one instance of CheckLegalMove in the constructor of the GameController class, which is represented by the checkLegal attribute. The CheckLegalMove manages moves that Actors are allowed to perform during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

CheckMill and Node


CheckMill
- millPosition: Map< int,ArrayList<ArrayList<int>>>
+ checkPossibleMill(ArrayList<Node>, int): void
CheckLegalMove
- adjacetPosition: Map<int,ArrayList<int>>
+ calculateLegalMove(Actor ,ArrayList<Node>):Map<int,ArrayList<Action>>
Token
- isMil: boolean
- id: String
+ updateStatus(): void
checks
holds
0...1
Logic.Location
Node
- position: Pair<int, int>
- contain: Token
- id: int
+ hasToken(): boolean
+ addToken(): void
+ removeToken(): void
+ allowableAction(ArrayList<Node>): ArrayList<Action>
1
24
checks

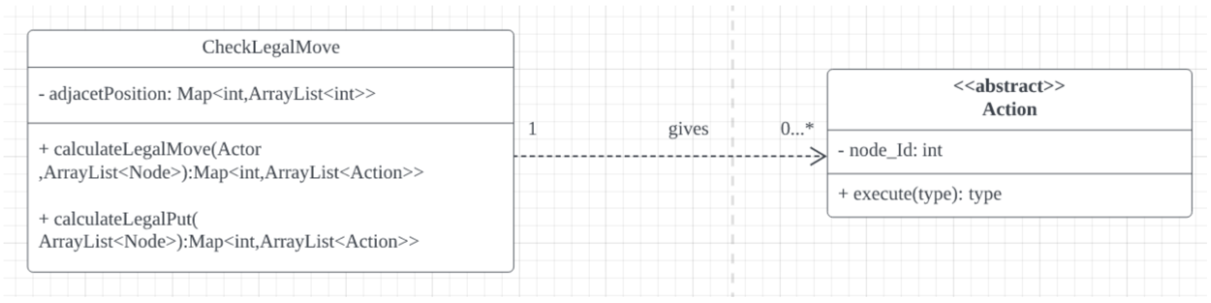
The multiplicity from CheckMill to Node class is 1 -> 24 because the CheckMill class reference 24 instances of Node class (ArrayList<Node>) is the list containing all 24 nodes in the game), which is passed from the Engine class as parameters, in the checkPossibleMill method.

CheckLegalMove and Node


CheckLegalMove
- adjacetPosition: Map<int,ArrayList<int>>
+ calculateLegalMove(Actor ,ArrayList<Node>):Map<int,ArrayList<Action>>
+ calculateLegalPut(ArrayList<Node>):Map<int,ArrayList<Action>>
holds
Logic.Location
Node
- position: Pair<int, int>
- contain: Token
- id: int
+ hasToken(): boolean
+ addToken(): void
+ removeToken(): void
+ allowableAction(ArrayList<Node>, ArrayList<Integer>): ArrayList<Action>
checks
1
24

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


CheckLegalMove
- adjacetPosition: Map<int,ArrayList<int>>
+ calculateLegalMove(Actor ,ArrayList<Node>):Map<int,ArrayList<Action>>
+ calculateLegalPut(ArrayList<Node>):Map<int,ArrayList<Action>>
1
gives
0...*
<<abstract>> Action
- node_Id: int
+ execute(type): type

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

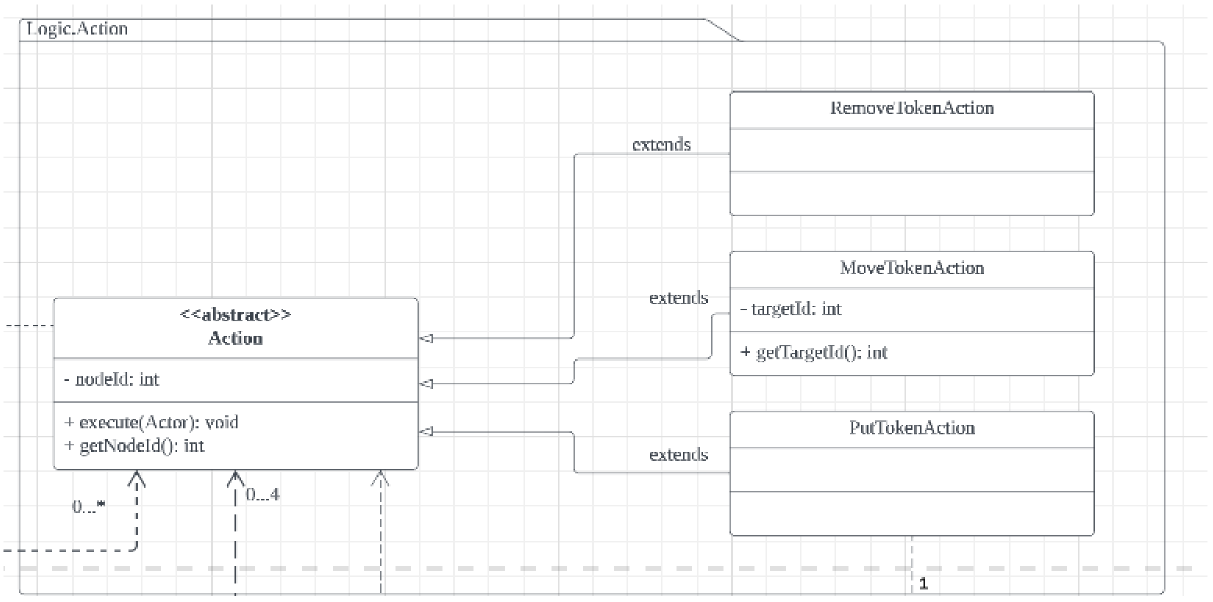
This design pattern was originally our initial choice for the design of the Abstract Action class, but it was then discarded since we believe that this approach would only cause complications for our implementation of the Nine Men Morris gameplay. This is because the set of possible actions that can 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

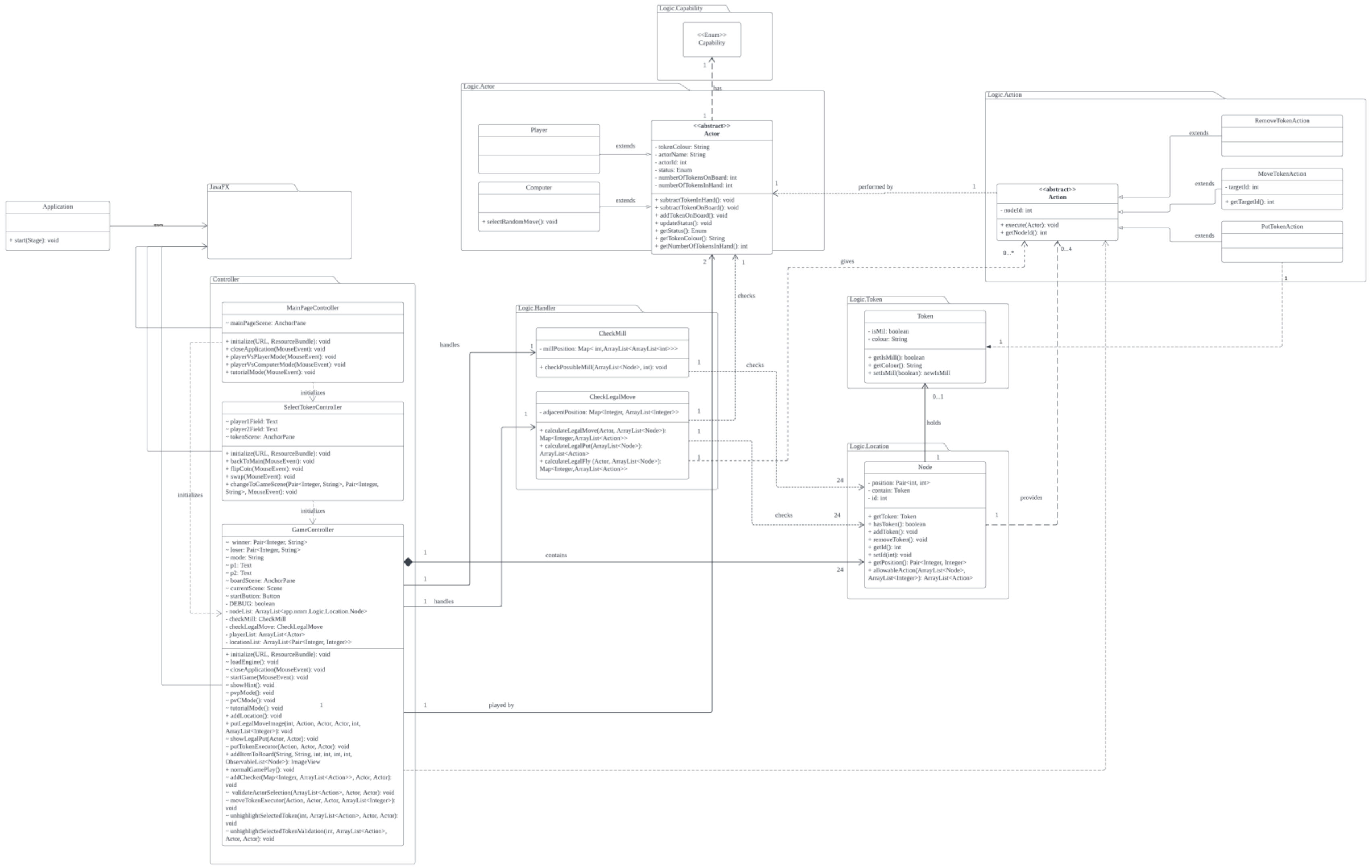


The updated version of the Action Abstract class from the last sprint.

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
has
Logic.Actor
Player
Computer
+ selectRandomMove(): void
extends
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
Application
+ start(Stage): void
uses
JavaFX
performed by
Logic.Action
<<abstract>>
Action
- nodeId: int
+ execute(Actor): void
+ getNodeId(): int
RemoveTokenAction
MoveTokenAction
- targetId: int
+ getTargetId(): int
PutTokenAction
extends
extends
extends
0...*
0...4
gives
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
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<app.nmm.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
~ pvpMode(): 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
~ addChecker(Map<Integer, ArrayList<Action>>, Actor, Actor): void
~ validateActorSelection(ArrayList<Action>, Actor, Actor): void
~ moveTokenExecutor(Action, Actor, Actor, ArrayList<Integer>): void
~ unhighlightSelectedToken(int, ArrayList<Action>, Actor, Actor): void
~ unhighlightSelectedTokenValidation(int, ArrayList<Action>, Actor, Actor): void
handles
handles
contains
played by
Logic.Handler
CheckMill
- millPosition: Map< int,ArrayList<ArrayList<int>>>
+ checkPossibleMill(ArrayList<Node>, int): void
CheckLegalMove
- adjacentPosition: 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
checks
checks
Logic.Token
Token
- isMil: boolean
- colour: String
+ getIsMill(): boolean
+ getColour(): String
+ setIsMill(boolean): newIsMill
0...1
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
24
24
24