FIT3077 Software Engineering: Architecture and Design

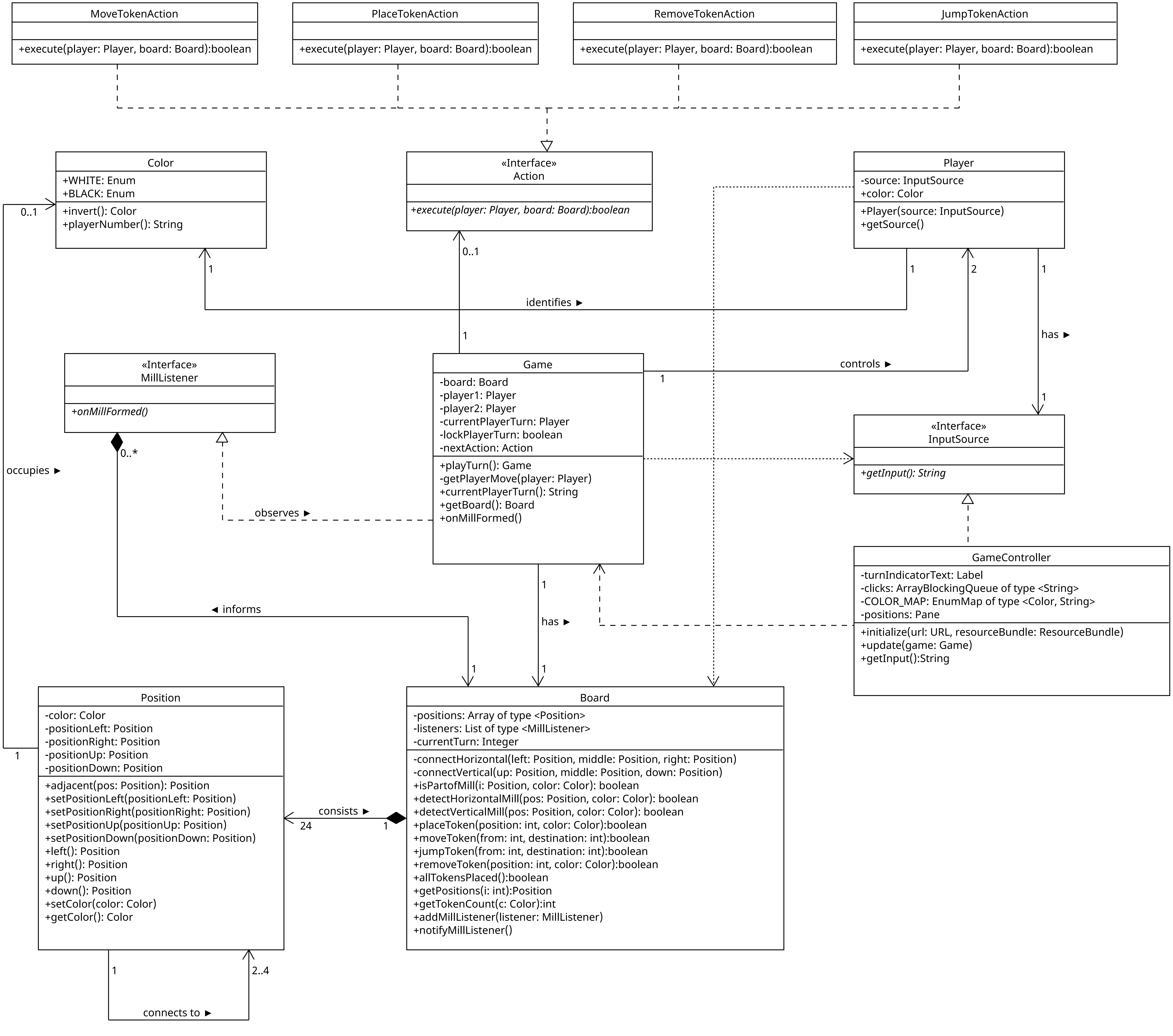
Assignment 2: Sprint Two

MA_Friday2pm_Team5

Table of Contents

Table of Contents 1
Architecture 2
Design Rationale 3
Game Model 3
Game Class 3
Actions Interface 3
MillListener Interface 4
Board & Position Class 4
Player 5
Controller and Game View 5

Architecture
Class Diagram



MoveTokenAction

+execute(player: Player, board: Board):boolean

PlaceTokenAction

+execute(player: Player, board: Board):boolean

RemoveTokenAction

+execute(player: Player, board: Board):boolean

JumpTokenAction

+execute(player: Player, board: Board):boolean

Color

+WHITE: Enum
+BLACK: Enum

+invert(): Color
+playerNumber(): String

«Interface»
Action

+execute(player: Player, board: Board):boolean

Player

-source: InputSource
+color: Color

+Player(source: InputSource)
+getSource()

«Interface»
MillListener

+onMillFormed()

Game

-board: Board
-player1: Player
-player2: Player
-currentPlayerTurn: Player
-lockPlayerTurn: boolean
-nextAction: Action

+playTurn(): Game
-getPlayerMove(player: Player)
+currentPlayerTurn(): String
+getBoard(): Board
+onMillFormed()

«Interface»
InputSource

+getInput(): String

GameController

-turnIndicatorText: Label
-clicks: ArrayBlockingQueue of type <String>
-COLOR_MAP: EnumMap of type <Color, String>
-positions: Pane

+initialize(url: URL, resourceBundle: ResourceBundle)
+update(game: Game)
+getInput():String

Position

-color: Color
-positionLeft: Position
-positionRight: Position
-positionUp: Position
-positionDown: Position

+adjacent(pos: Position): Position
+setPositionLeft(positionLeft: Position)
+setPositionRight(positionRight: Position)
+setPositionUp(positionUp: Position)
+setPositionDown(positionDown: Position)
+left(): Position
+right(): Position
+up(): Position
+down(): Position
+setColor(color: Color)
+getColor(): Color

Board

-positions: Array of type <Position>
-listeners: List of type <MillListener>
-currentTurn: Integer

-connectHorizontal(left: Position, middle: Position, right: Position)
-connectVertical(up: Position, middle: Position, down: Position)
+isPartofMill(i: Position, color: Color): boolean
+detectHorizontalMill(pos: Position, color: Color): boolean
+detectVerticalMill(pos: Position, color: Color): boolean
+placeToken(position: int, color: Color):boolean
+moveToken(from: int, destination: int):boolean
+jumpToken(from: int, destination: int):boolean
+removeToken(position: int, color: Color):boolean
+allTokensPlaced():boolean
+getPositions(i: int):Position
+getTokenCount(c: Color):int
+addMillListener(listener: MillListener)
+notifyMillListener()

Design Rationale

The software architecture is designed to cover the basic requirements of 9 Man Morris (9MM) game and to provide a maintainable and scalable platform for future extensions of the advanced requirement. Holistically, the software uses the traditional model-view-controller architectural pattern as a foundation of how the classes are designed and modelled after. The model-view-controller (MVC) pattern is a straightforward pattern of choice to the team as it provides a clear decoupling between the player input, display and game model which is a suitable model for the requirements of the game.

Game Model

Game Class

The game model is primarily the Game class and its associated variables. The main loop of the Game class (playTurn) acquires the next Action which are instructions to change the state triggered by the player. Whenever an Action is executed, the method will return the Game class itself for the View component of the Game to update the display to the user. The reason for this approach is that for the basic 9MM requirements the Game class does not yet have to be an *active MVC* as the game state only needs to be computed after a player's input. In the future, the Game class may need to be an active MVC when there are more classes interested with the state of the game but the team decided to keep things simple for this stage of the development. As this class is a composition of other game components, it has an association relationship with the Board class, a multiplicity of 2 with Player class despite having 3 instance variables as currentPlayerTurn references to one of the same players.

Actions Interface

The Action interface is the unit of transaction where the player interacts with the board. Fundamentally, the Player will create specific inputs that will be read by Actions at various stages of the game and the object itself will check the validity of the transaction before it is executed. As such, the 4 main actions the players can interact with the board related with the game, i.e. PalaceTokenAction, JumpTokenAction, MoveTokenAction and RemoveTokenAction will each implement the logic of checking the Player input with the current state of the Board. Action is also an interface instead of an abstract class as there are not any notable instance variables that will be useful for the implementation and maintains the flexibility of multiple interfaces for implemented objects in the future.

The motivation behind encapsulating the player's interaction as Action is inspired by the Command Pattern. One direct benefit of the pattern is the decoupling of the game model from the controller, where the game model may directly retrieve commands from the controller without this approach. Through this decoupling step, Action sources have the interaction flexibility with the game model by implementing an InputSource that dictates where the commands are retrieved. This will be useful in implementing computer moves for the game where the game model and action is unaffected by a different InputSource since we free ourselves from the constraint of only the GameController being allowed to provide input. As Actions need to change the specific state of the Board, they have a dependency relationship.

To summarize, the Game class creates a specific Action based on the Game State, and that Action will read the next input by the Player's InputSource and verify if it is a correct move.

MillListener Interface

By default, our Game model alternates between two players after an action is executed. However, in 9MM the player is able to remove one of the other player's tokens under RemoveTokenAction, as such it would miss the turn under the current model as another prior action by the player is executed.

The most straightforward solution is to have Board class inform Game that a mill is formed, however that would violate the acyclic dependency principle as Board and Game's relationship forms a cycle. For a more scalable option, the team went with the observer pattern to introduce a MillListener interface that will notify its implementer that a mill is formed. Board will have a list of MillListeners (0..*) that will be notified when a Mill is formed, and Game implements the MillListener interface to give the Player an extra turn to remove the token. The list of MillListeners is part of the Board and a composition relationship as Mill can only be formed within the Board. The advantages of this approach is not only scalable, the interface is also not constrained to only inside the game model, for example we can notify the UI a Mill is formed and do some interesting visualizations.

However, the drawback of this approach for the Game class is that it solidifies the relationship between Game and Actions from dependency to association, which has the multiplicity (0..1) depending on if there is a next action awaiting execution.

Board & Position Class

The Board class is part of the composition to Game class. However, the difference between a Board and Game is that Board serves as a data structure to store the state of the Positions after executing some Action, whereas Game provides the general flow and encapsulates the entire 9MM game. The Board class also provides methods to manipulate the state in which it returns a boolean indicating a success or failure. The team originally wanted to throw exceptions but went on with the simpler approach of returning true or false indicating the validity of the move.

A Position is a unit that collectively forms the Board and they represent a single placeable location in 9MM. Hence, the Board has exactly 24 instances of the Position object representing a location each, and has a composition relationship with Position as they make up the game board. Each position is also connected with other Positions just as drawn on a 9MM Board (2..4), and they are each saved according to the relative direction to the appropriate variable. This is to assist identifying Mill with connected positions more easily.

On the other hand, a Color is an enum that represents White or Black. Each Position has an association with a Color variable to indicate if it is currently occupied by White, Black token or null if unoccupied. The decision to go with using Color to “occupy” a position instead of having a distinct Token class is because there were not enough responsibilities to justify the existence of the Token class other than occupying a Position. The Color is also useful in identifying the player and calculating if the player move is valid in the Board and Action class.

Player

This class represents the two players of 9MM where each has an InputSource that are the same for the basic 9MM and are identified by their Color. The player has an InputSource to decouple themselves to a single type of controller that is able to generate an input just as the dependency inversion principle. The InputSource represents where the Player command is generated, and provides how the player wishes to interact with the game.

Controller and Game View

The controller for the user to interact with the game is the GameController class. The class implements the InputSource interface to have a unified method of retrieving the desired input position of the player. Upon clicking on a position of the board, the controller “queues” the input and Action will read the input from the queue. The Game class will then change its state based on input and the controller will use the returned Game object to update the display. Ideally, the View component should be its own class but due to constraints of the framework it is represented by a single “update” method inside the Controller class.

It is worth noting that JavaFX uses FXML (similar to HTML) and CSS for its GUI hence the modeling for display was not properly reflected in the class diagram. The framework uses annotations (@FXML) to directly inject the View attributes into the controller class which may have caused a harder distinction between them. Nonetheless, the responsibilities are still clearly differentiated from the methods where the display is told how to manipulate the appearance by a single method (update) in Controller. Ultimately, the View still relies on a markup language to render the display to the user, and the View still uses the computed Game state to render the next frame after being informed by the Controller. It is a limitation of class diagrams not being able to show the full picture of View’s components. The flaw of this approach is that it is a passive MVC which requires GameController to initiate the entire sequence, but we feel this approach is sufficient for the basic 9MM requirements.