Sprint 2
FIT 3077 Group 29
Tycl's inc. ©
Terry Truong 32466870
Cynthian Thai
Leon Ma 31449840
Yuhan Zhou 31468748

Sprint 2 Rationale

Token and Position

We debated whether Token should be an attribute or a class as we are using JavaFX we needed to figure out the optimal way to represent a token. One alternative was to separate the token into its own class and know its own position rather than as an attribute in Player or Position depending on where it is in the game. By separating a token into its own class, we ensure encapsulation as all logic and behaviour associated is kept in one place and allows for expandability for future use of the token. Furthermore, by doing this, the code is more modular with Token having its own single responsibility and thus it is easier to maintain. We can also maintain reusability as multiple classes will need access to tokens where its related behaviour and logic is defined. This reduces re-coding behaviours again within other classes. A disadvantage of this alternative is the token having minimal responsibilities and may become a data holder class.

Another alternative that was considered was the Position has a Token attribute that can be checked for if a token has been placed on it. For our use of JavaFX, this allows us to click the token and next position rather than drag to move tokens. This can help maintain simplicity since Token does not have much behavioural logic. However, the disadvantages of this is the token attribute being used across multiple classes (Board, Move, etc.) can create unnecessary dependencies to Position, this makes the code harder to maintain, test and debug. This also does not follow Open-Closed Principle as if there becomes additional functionality that relies on Token, the Position class will need to be modified which can also make the Position class have too many responsibilities, breaking the Single responsibility principle.

Ultimately, we separated the Token to a class instead of keeping it as an attribute for our design. By doing so, it adheres to Object Oriented Design, following the principles of single responsibility and open-close principle with encapsulation, making the overall system much easier to maintain and extend.

Moves, MoveAction and Player

Another issue we debated was whether player moving a token should be a method within Player class or separated into a class. The first alternative would be having a player calculate their possible token move in a method in Player. The pros of doing this reduces dependencies as the Player takes their turn with their known move. The disadvantage of this is it breaks open-closed principle as if more moves are added, the Player code would need to be modified rather than extending the code. This is also breaking single responsibility as the Player class can also do other actions such as undo or showing a hint that do not involve a move on the board which can become difficult to maintain and to extend without modifying in the future.

Another alternative was Move being separated into an abstract class rather than a method. Doing this follows the open-closed principle as the abstract Move class can be extended without having to modify the existing Player code, ensuring encapsulation and modularity. This also allows the functionality of a game move to be separated from undo and hint actions making it easier to test and reuse in future functionalities.

Ultimately, defining a separate class for Move was chosen as the final design as it permits behaviour and information relevant to a move that alters the state of the board to be segregated and confined in one location without risk of Player becoming a God class. This will make it easy to modify and enhance its behaviour in the future.

Relationships, Inheritance and Cardinalities

There were two distinct inheritance classes that have been decided upon for our code architecture design, this includes the Action class which is inherited by HintAction, UndoAction, and MoveAction child classes. The other parent class is the Move abstract class which is inherited by the subclasses Slide, Milling, Place, and Hop. The reason for an Action parent class allows us to minimise repeating code that would apply to all the classes that control their respective actions which the player can take, with this being showing hints, undoing their previous moves, or moving a piece.

A key relationship present in the class diagram is the inheritance of the Move abstract class with the different types of movement possibilities for each token. This is important in our class diagram as the inheritance from the Move class allows for the “Mill”, “Slide”, “Hop” and “Place” classes to all inherit their key movement from the Move class whilst also being able to override methods changing it to their liking. This makes coding much more flexible, efficient and adheres to Liskov’s substitution principle since each will be allowed to substitute for the parent Move adheres to Dependency Inversion since MoveAction, a high-level class, will rely on the abstract Move class avoiding dependencies on concrete classes. On the contrary, if the relationships between “Slide”, “Hop” and “Place” were compositions. Large amounts of duplicated code are produced since overlapping movement behaviour will need to be defined each time within all the classes. Overtime, the code can become extremely difficult to modify and extend, breaking Open-closed principle, as changes will need to be made across multiple places.

These two abstractions also allow us to generalise the roles of the subclasses with their parent classes with it being Action generalising all the actions a player can choose to take, and Move generalising all the different types of moves a player can utilise at specific points into the game. This helps to follow the Dependency Inversion principle as the classes will rely on abstractions rather than concrete classes.

Another important relationship present in our class diagrams is the direct association the Token class has with the Position class. The association is present due to the Token class needing to know its position on the board. The token class can either have a position or be off the board, either in the player’s hand if it has not yet been placed or in the opponent's hand, if it has been captured. This allows Token to know about its position that will be updated each time move is called. This is represented by the cardinality pairs for Token and Position, with 1 Token being “on” 0 or 1 Position, indicating that Token can either have one Position or no Position. The Position is a set spot on the board and is all instantiated on board creation, which means that at the start of the game, no Tokens will exist on the board therefore meaning no Positions will have a Token.

The set of cardinalities between the Player and Token relationship can be explained by a player having 0 to 9 tokens tied to their instance and each token has only one player it is related to as the tokens are tied to one team. We arrived at the set of cardinality as each player plays with a total of nine tokens, which may diminish over the span of the game as tokens can be captured/removed from the game when a mill has been created. The token however is assigned a specific colour which indicates with one of the two players it is owned by, with this, it ensures that each token only relates to one instance of a player.

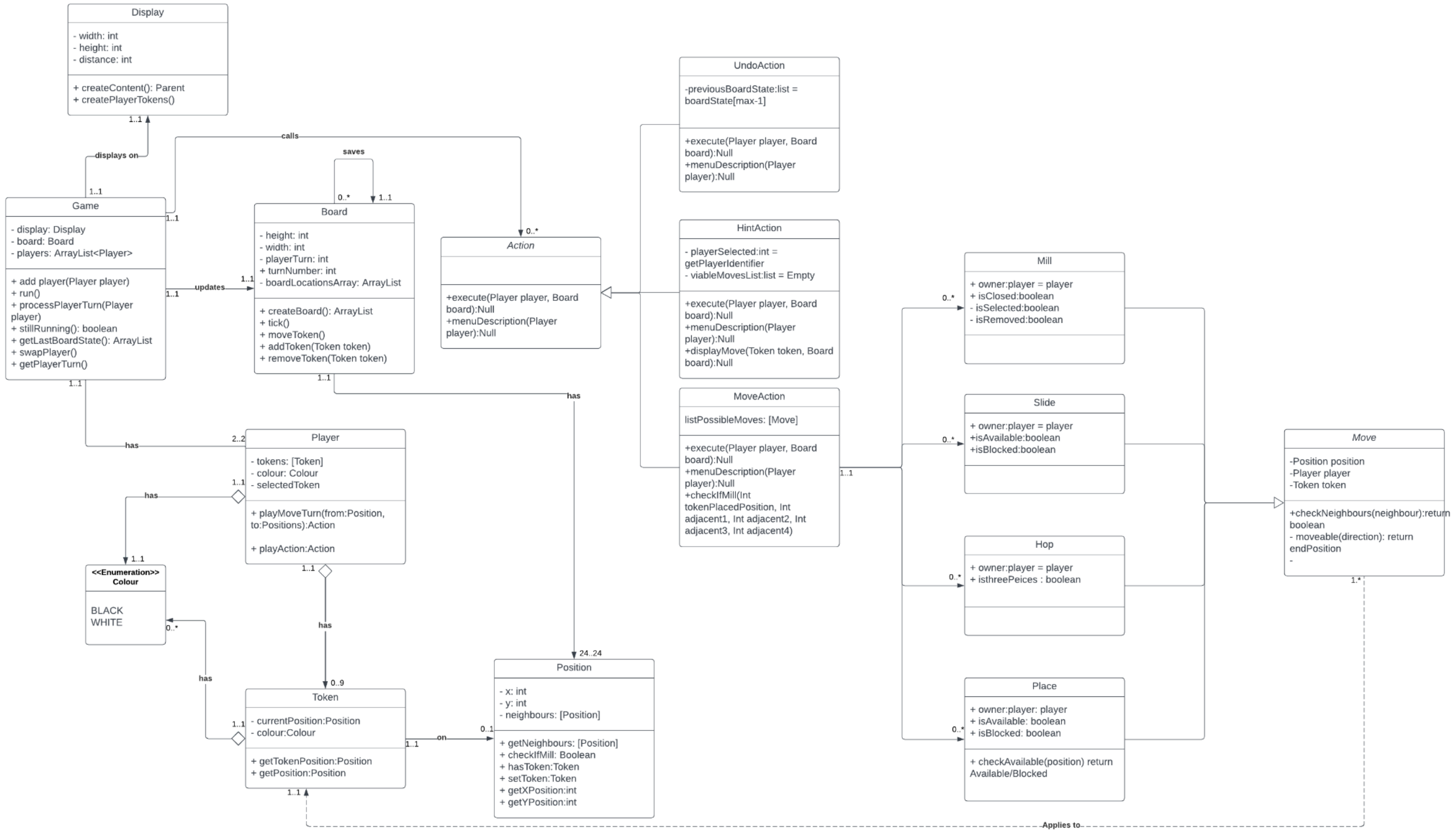
Software Architecture Design Pattern

One of the prominent design patterns that has been applied in this system is Chain of responsibility. Following this designation pattern, it allows a particular order of operation to be executed but also avoids coupling between the sender (Game) from the receiver(Move) and allows the intermediate classes to handle a particular action. This also allows the code to become more modifiable and extendable as if the order of execution were to change, little changes are needed to be changed to the Game class and the Move class as order of execution are handled in between and thus adhering to open/closed principles of Object Oriented Design. Furthermore, this also allows moveAction, Mill, Hop, Slide and Move to act as one set of classes and abstract away the complexity of handling the move action from the sender, Game class.

Other feasible alternatives were also considered to deal with this subsystem. Implementing different player actions within the Action class itself was one of them. Implementing this design instead will create unnecessary dependencies and communication between other components. By containing all action logic within the Action class, this will result in Mill, Hop, Slide classes inheriting this class instead of moveAction and now these classes will also inherit hintAction and undoAction which is irrelevant to Mill, Hop and Slide and thus breaks single responsibility principle.

Another alternative is using the Observer pattern instead where the observers (Game, board, Token, etc.) subscribes to Move. When the Move class conducts a certain move in this solution, the subscribers are immediately updated. However, doing so will result in a huge number of unnecessary associations between Move and the other classes. By retaining a Game instance within Move, all classes that rely on Game will also rely on Move. A large number of dependents makes the code difficult to debug since faults cannot be isolated, but it also causes errors to propagate through these dependencies.

Class Diagram



Display
- width: int
- height: int
- distance: int
+ createContent(): Parent
+ createPlayerTokens()
displays on
1..1
1..1
calls
saves
0..*
1..1
Game
- display: Display
- board: Board
- players: ArrayList<Player>
+ add player(Player player)
+ run()
+ processPlayerTurn(Player player)
+ stillRunning(): boolean
+ getLastBoardState(): ArrayList
+ swapPlayer()
+ getPlayerTurn()
updates
1..1
1..1
Board
- height: int
- width: int
- playerTurn: int
+ turnNumber: int
- boardLocationsArray: ArrayList
+ createBoard(): ArrayList
+ tick()
+ moveToken()
+ addToken(Token token)
+ removeToken(Token token)
0..*
Action
+execute(Player player, Board board):Null
+menuDescription(Player player):Null
UndoAction
-previousBoardState:list = boardState[max-1]
+execute(Player player, Board board):Null
+menuDescription(Player player):Null
HintAction
- playerSelected:int = getPlayerIdentifier
- viableMovesList:list = Empty
+execute(Player player, Board board):Null
+menuDescription(Player player):Null
+displayMove(Token token, Board board):Null
MoveAction
listPossibleMoves: [Move]
+execute(Player player, Board board):Null
+menuDescription(Player player):Null
+checkIfMill(Int tokenPlacedPosition, Int adjacent1, Int adjacent2, Int adjacent3, Int adjacent4)
1..1
Mill
+ owner:player = player
+ isClosed:boolean
- isSelected:boolean
- isRemoved:boolean
0..*
Slide
+ owner:player = player
+isAvailable:boolean
+isBlocked:boolean
0..*
Hop
+ owner:player = player
+ isthreePeices : boolean
0..*
Place
+ owner:player: player
+ isAvailable: boolean
+ isBlocked: boolean
+ checkAvailable(position) return Available/Blocked
0..*
Move
-Position position
-Player player
-Token token
+checkNeighbours(neighbour):return boolean
- moveable(direction): return endPosition
-
1.*
1..1
has
1..1
has
2..2
Player
- tokens: [Token]
- colour: Colour
- selectedToken
+ playMoveTurn(from:Position, to:Positions):Action
+ playAction:Action
1..1
has
1..1
<<Enumeration>> Colour
BLACK
WHITE
0..*
has
1..1
has
0..9
Token
- currentPosition:Position
- colour:Colour
+ getTokenPosition:Position
+ getPosition:Position
1..1
on
1..1
0..1
24..24
Position
- x: int
- y: int
- neighbours: [Position]
+ getNeighbours: [Position]
+ checkIfMill: Boolean
+ hasToken:Token
+ setToken:Token
+ getXPosition:int
+ getYPosition:int
1..1
Applies to