FIT3077: SOFTWARE ENGINEERING: ARCHITECTURE AND DESIGN

Sprint 2 (25%)

Design Rationale

MONASH University

Done By

Shillong Xiao (Norman) (30257263)

Sok Ear (32442688)

Kisara Batugedara (30912539)

Faculty of Information Technology

Note for the project

The working game prototype can be seen under **FX branch**.

Rationale

Key classes

Decision 1: MoveAction entity is to move a piece from one position to another. Therefore, should it be created into a method or a class?

<table>
<tr><td>Alternative 1

MoveAction is made into a class</td><td>Alternative 2

MoveAction is created into a method that is executed by Game Class.</td></tr>
<tr><td>Advantages

- This creates encapsulation for move functionality which is isolated from the rest of the code. If the move function were to change in the future, this can be done so easily without heavily impacting the rest of the system. This promotes extendibility.
- MoveAction class will have a well-defined responsibility which conforms to the single responsibility principle.
- It will be useful if there is an undo move functionality to implement in the future. This is as the detail of each move is stored in a class instance.

Disadvantages
- It will increase the level of dependency in the code. MoveAction class will require information such as Piece, Player, Position and Board to be able to implement a move. This can increase the complexity of the code.
- MoveAction class will need to be executed by Game class. Therefore, this can lead to feature envy of the Game class to MoveAction class, making the 2 classes inappropriately dependent.</td><td>Advantages:
- It reduces dependency in the code since everything is done in Game class. Game class already has dependency to Player, Position, Token and Board classes. All of which are required to implement a move.
- For a small project with minimal future extension requirements, doing this can simplify the implementation process by removing the need to create another class.

Disadvantages
- This increases the responsibility of the Game class, which increases the risk of it being a god class. This is a code smell which can lead to lower maintainability.</td></tr>
<tr><td colspan="2">Decision</td></tr>
</table>

After careful consideration, alternative 1 is selected. One of our key criteria is extendibility. Having a MoveAction be a class, it can make the code more encapsulated and isolated from other functionalities. Since the requirements of future sprints are unknown, this will increase flexibility in the codebase which can help drive the success in future sprints.

Decision 2: The Hint button must highlight the specific positions that the user can place their tokens at when they're unsure of what to do next. Therefore, should Hint be implemented as a class or just a method inside the board class that executes the functionality?

Alternative 1 Hint is made into a class that handles all functionalities regarding hints.	**Alternative 2** Hint is made into a method inside the board class that gets executed when the user clicks on the hint button
Advantages - It follows the Open/Closed Principle, where the class can be open for extension and closed for modification, if there were to be added functionalities in terms of hints for the game. - It has one single responsibility which is highlighting all the positions that the player can move their token into. Therefore it conforms to the Single Responsibility Principle - Less prone to errors as it conforms to encapsulation. All data related to the hints are within a single entity. - If the game was to have more types of hints available to the player, the design can easily be extended using inheritance by creating sub-classes of hints. **Disadvantages** - Making it a class can lead to decreased performance - Making a class can introduce more complexity - It will require more development time as it requires integration with other classes.	**Advantages:** - It ensures simplicity as it promotes easy readability and maintainability. - It is more efficient **Disadvantages** - Hint isn't the responsibility of the board class. Therefore, having the hint functionality inside the board class can violate the single responsibility principle.

Decision After careful consideration, alternative 1 has been selected. This is because it will promote maintainability by having a separate HintAction module that can take care of hint action execution without affecting other parts of the code.	

Key relationships

Decision 1: What type of relationship should Board class have to Position class?

Alternative 1 Board has an aggregation relationship to Position	**Alternative 2** Board has an composition relationship to Position
Advantages - It allows Position to exist outside the Board. Therefore, it can increase flexibility in the code if there were to be further change of requirements - It provides a way to represent situations when a token is outside the board (when it has not been placed yet or when it has been removed). - It encourages reusability of Position class since there is less restriction. **Disadvantages** - Position can be defined inappropriately leading one Position to belong to two different Boards. This can crash the game.	**Advantages:** - Composition relationship enforces that position must exist in Board therefore it reduces the likelihood of misuse of the class. **Disadvantages** - There is an association relationship between Token → Position. When a token is not on the board, its position attribute must be represented as "null" since position cannot exist outside a Board. This use of null values can introduce higher complexity in handling and increases the risk of errors during game execution. Errors related to null pointer exceptions are generally more difficult to trace and can therefore make maintenance more challenging.
Decision Upon careful consideration, alternative 1 is adopted. This is because it increases flexibility of the Position class, making it a more viable option. Furthermore, alternative 2 is likely not a good method due to the handling of “null” type which can make the system more error-prone.	

Decision 2: What type of relationship should MoveAction class have to Board class?

Alternative 1 MoveAction has a dependency to Board	**Alternative 2** Move action has an association relationship to the Board.
Advantages - This will allow MoveAction to utilise certain attributes and functions from the Board class which will simplify the implementation of the class. - Encourages a more modular design relative to an association relationship. **Disadvantages** - Dependency is hard to test. This is because a mock instance of Board will need to be created during testing.	**Advantages:** - Association often simplifies the codebase making it easier to use and understand. **Disadvantages** - It can create unnecessary tight coupling between the classes, making it harder to maintain and making refactoring more costly.
Decision Upon careful consideration, alternative 1 is chosen due to dependency having lower coupling than association. This can help keep the code relatively modular which can improve maintainability.	

Inheritance

Decision 1

There are two game modes: Tutorial Game Mode and Pvp Game Mode. In the current project, the tutorial game mode will work as a slide show showing different game situations rather than allowing the users to play the game with a computer. Should there be a Game abstract class where PvpGame and TutorialGame will extend from?

Alternative 1 Game is an Abstract class. TutorialGame and PvpGame will extend from it.	**Alternative 2** Don't have an abstract class. Game class will represent pvp game mode. TutorialGame will be a separate class not related to Game class.
Advantages - It endorses open-closed principles. If there were to be more game modes created, it will allow easy	**Advantages:** - No inheritance allows for a simpler code design which improves readability.

extension. The implementation process will be quicker since only the specific functions required for the new game mode need to be defined. It enhances reusability of the code and aligns with DRY (Don't repeat yourself principle). **Disadvantages** - There is little similarity between PVP game mode and Tutorial game mode. Therefore, Game abstract class will serve very minimal purpose.	- Game and Tutorial Game can behave differently rather than conforming to a parent class. This increases flexibility. - It can be easier to test due to low dependency between classes since each game mode class is separate. **Disadvantages** - It may lead to repetition of code of some functionalities if there is an overlap between Game and TutorialGame. - It decreases encapsulation. Other classes will need to interact with Game class and TutorialGame class directly without another layer of abstraction. This can lead to increased risk of inappropriate intimacy between classes, making it harder to maintain.
Decision Upon careful consideration, alternative 2 is adopted. Since the current design of the game does not share many similarities between the tutorial game mode and PvP game, abstracting the game class may not provide significant benefits. By creating separate and unrelated classes for the game and tutorial game modes, it can increase flexibility and make it easier to modify and extend each mode independently.	

Decision 2

In 9 men's morris, there are 3 possible types of moves which are initial placing move, sliding move and jump move. Should there be a MoveAction abstract class?

Alternative 1	**Alternative 2**
MoveAction is an abstract class. It will have InitialPlacingMoveAction, SlidingMoveAction and JumpMoveAction extending from it.	MoveAction is a regular class. It will have multiple methods such as initialPlacingMove(), slidingMove() and jumpMove() instead.
Advantages - Similarly to decision 1, it endorses open-closed principles. This can allow other types of moves to be implemented easily if it were to be extended in the future.	**Advantages:** - It allows flexibility when new types of moves are implemented. The new class will not be bound by any abstraction. - It opens the possibility of

- By using the MoveAction class abstraction, Game can conveniently execute all types of moves. **Disadvantages** - This increases the complexity of the system which can lead to the risk of over-engineering. If there is little need for extension in the system, it will waste implementation time. - There is an increased risk of violating the substitution principle in the future, as new move classes may not perform the same functionality as their parent classes. This can result in high refactoring costs if the parent class needs refactoring to accommodate for this.	MoveAction class determining what type of moves to be implemented due the current stage of the game. Doing so can reduce the load of the Game class because of the game, thereby endorsing a more balanced responsibility of classes. This is feasible because MoveAction already has a dependency to board to determine the stage of the game. **Disadvantages** - MoveAction class will require constant modification if new types of moves are added. This can make the class hard to manage and maintain.
Decision Upon careful consideration, alternative 1 is selected. The use of abstraction here is reasonable due to each type of move behaving similarly. Furthermore, it increases extendibility which will be important for later sprints.	

Cardinalities

Decision 1

Cardinality Board has an association with Token. Cardinality is 1 → 0….18
Explanation - This captures the reality of the game. This is because the Board starts off empty, therefore it will have 0 tokens. When not only board, a token will be placed in a position outside the board. The maximum number of tokens on the board is 18. For the current design, there are no other feasible alternatives.

Decision 2

Cardinality Player has an association with Token

Cardinality is 1 → 9.
Explanation: The number of tokens for each player is fixed at nine from the beginning of the game until the end. If one of the player's tokens is removed due to a mill move by the opponent, the token will not be taken away from the player, but instead, it will be removed from the board and will have its PLAYABLE capability removed. In turn, the token can no longer be played. This is the only feasible cardinality that is consistent with the current design.

Design pattern

Decision 1

Currently, only one game will need to run at a time. Should Game class be a singleton?

Make Game class a singleton
Advantages - In the current game requirement, where only one game can exist at a time, the use of a Singleton design pattern can ensure that only one instance of the Game class is created. This can reduce the risk of misuse of the Game class by preventing multiple instances from being created and potentially causing crashes and unexpected behaviour. **Disadvantages** - This limits the ability for multiple games to be run and stored in the future if it becomes a requirement. This limits flexibility. - Singleton classes can decrease testability since they can not be mocked easily. .
Decision

Upon careful consideration, we have decided to make Game class a regular class. Although currently only one instance of Game is needed, future requirements may require that multiple instances of game classes be stored. Therefore, due to reduced flexibility, a singleton class for Game is not considered.

Decision 2

Since MoveAction is an abstract class, in order to implement SlideAction, JumpAction and PlaceAction, should we make MoveAction an abstract Factory?

Make MoveAction an abstract factory

Advantages

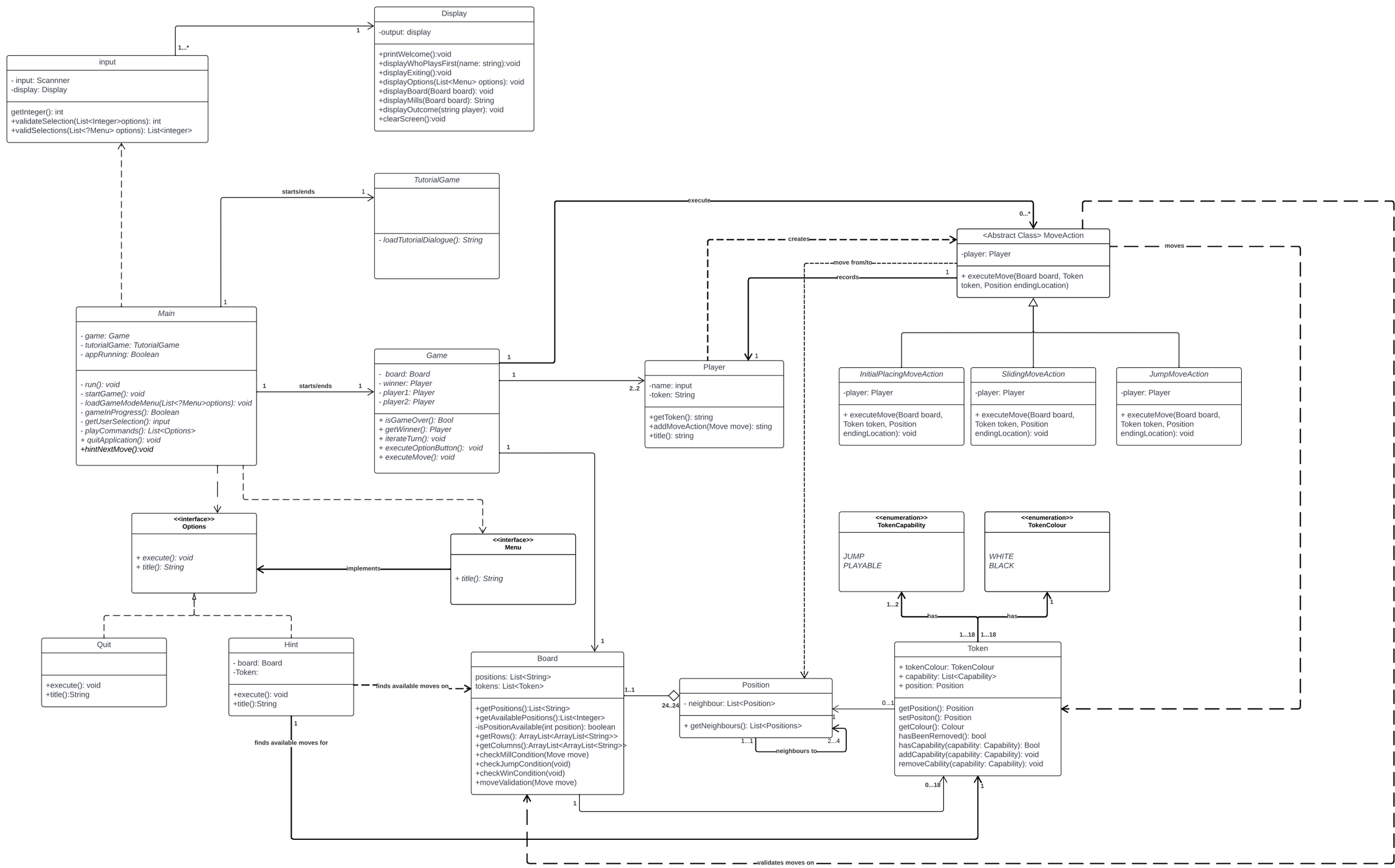
- This design pattern encapsulates the creation of objects, providing a clear separation of concerns between the client code and creation of objects. This makes it easier to change the way objects are created.
- This pattern makes it easier to create families of related objects ensuring flexibility and adaptability.
- This pattern promotes consistency in object creation ensuring all related objects are created in a consistent manner
- This pattern makes it easier to test the code because it allows for the creation of mock objects that can be used for testing.

Disadvantages

- This design pattern can lead to additional complexity in our code as it could end up creating large families of related objects.
- Due to its complexity, it may require more time in terms of development.
- It is very limited in terms of extensibility as it requires significant number of changes to the code if new types of objects were to be created
- Since additional abstraction layers exist in this design pattern, it can lead to decreased performances in some cases.

Decision

Upon careful consideration, we have decided to make MoveAction class as an abstract factory. It is easier for us in the future to call the specific type of action in order to implement on tokens. Although the complexity of connecting the methods with the other classes is high, it would help us troubleshooting the errors and recording the action history for reference.


Display
-output: display
+printWelcome():void
+displayWhoPlaysFirst(name: string):void
+displayExiting():void
+displayOptions(List<Menu> options): void
+displayBoard(Board board): void
+displayMills(Board board): String
+displayOutcome(string player): void
+clearScreen():void
input
- input: Scannner
-display: Display
getInteger(): int
+validateSelection(List<Integer>options): int
+validSelections(List<?Menu> options): List<integer>
1
1...*
TutorialGame
- loadTutorialDialogue(): String
starts/ends
Main
- game: Game
- tutorialGame: TutorialGame
- appRunning: Boolean
- run(): void
- startGame(): void
- loadGameModeMenu(List<?Menu>options): void
- gameInProgress(): Boolean
- getUserSelection(): input
- playCommands(): List<Options>
+ quitApplication(): void
+hintNextMove():void
starts/ends
Game
- board: Board
- winner: Player
- player1: Player
- player2: Player
+ isGameOver(): Bool
+ getWinner(): Player
+ iterateTurn(): void
+ executeOptionButton(): void
+ executeMove(): void
execute
0...*
creates
move from/to
records
<Abstract Class> MoveAction
-player: Player
+ executeMove(Board board, Token token, Position endingLocation)
moves
Player
-name: input
-token: String
+getToken(): string
+addMoveAction(Move move): sting
+title(): string
2..2
InitialPlacingMoveAction
-player: Player
+ executeMove(Board board, Token token, Position endingLocation): void
SlidingMoveAction
-player: Player
+ executeMove(Board board, Token token, Position endingLocation): void
JumpMoveAction
-player: Player
+ executeMove(Board board, Token token, Position endingLocation): void
<<interface>>
Options
+ execute(): void
+ title(): String
implements
<<interface>>
Menu
+ title(): String
<<enumeration>>
TokenCapability
JUMP
PLAYABLE
<<enumeration>>
TokenColour
WHITE
BLACK
has
has
1...2
1...18
1...18
Quit
+execute(): void
+title():String
Hint
- board: Board
-Token:
+execute(): void
+title():String
finds available moves on
finds available moves for
Board
positions: List<String>
tokens: List<Token>
+getPositions():List<String>
+getAvailablePositions():List<Integer>
-isPositionAvailable(int position): boolean
+getRows(): ArrayList<ArrayList<String>>
+getColumns():ArrayList<ArrayList<String>>
+checkMillCondition(Move move)
+checkJumpCondition(void)
+checkWinCondition(void)
+moveValidation(Move move)
1..1
24..24
Position
- neighbour: List<Position>
+ getNeighbours(): List<Positions>
1...1
2...4
neighbours to
0...1
Token
+ tokenColour: TokenColour
+ capability: List<Capability>
+ position: Position
getPosition(): Position
setPositon(): Position
getColour(): Colour
hasBeenRemoved(): bool
hasCapability(capability: Capability): Bool
addCapability(capability: Capability): void
removeCability(capability: Capability): void
0...18
validates moves on