MONASH University

ENGINEERING

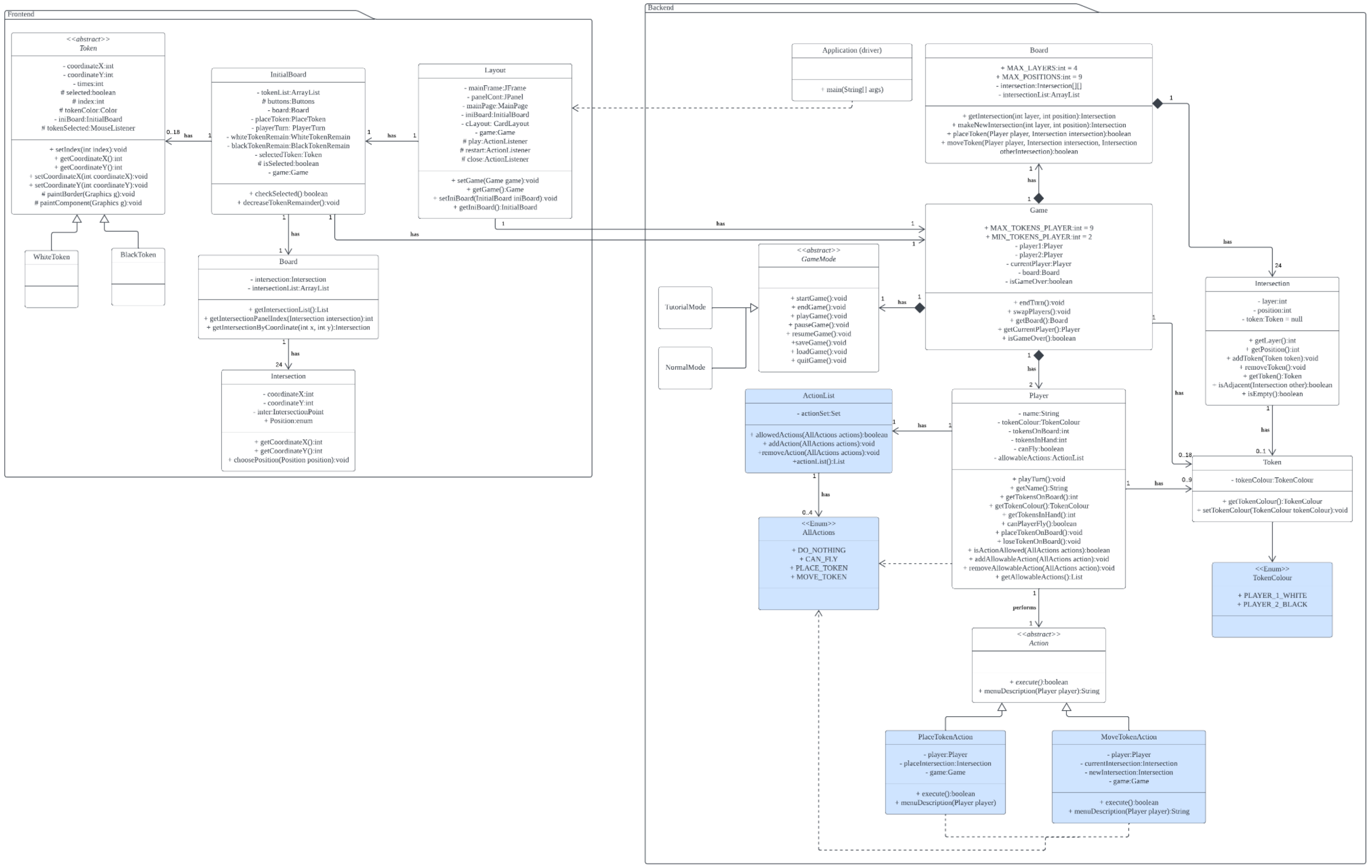
SPRINT 2

TABLE OF CONTENTS

1. Architecture and Design Rationales **2**
1.1. Class Diagram (changes are highlighted in blue) *Clearer version can be viewed through the link in appendix 2
1.2. Design Rationales 3
Backend 3
Frontend (every classes mentioned below are added in Sprint 2) 5
1.3. Discarded Alternatives 6
2. APPENDIX **7**
Class Diagram 7

1. Architecture and Design Rationales

1.1. Class Diagram (changes are highlighted in blue) *Clearer version can be viewed through the link in appendix


Frontend
<<abstract>> Token
- coordinateX:int
- coordinateY:int
- times:int
# selected:boolean
# index:int
# tokenColor:Color
- iniBoard:InitialBoard
# tokenSelected:MouseListener
+ setIndex(int index):void
+ getCoordinateX():int
+ getCoordinateY():int
+ setCoordinateX(int coordinateX):void
+ setCoordinateY(int coordinateY):void
# paintBorder(Graphics g):void
# paintComponent(Graphics g):void
WhiteToken
BlackToken
InitialBoard
- tokenList:ArrayList
# buttons:Buttons
- board:Board
- placeToken:PlaceToken
- playerTurn: PlayerTurn
- whiteTokenRemain:WhiteTokenRemain
- blackTokenRemain:BlackTokenRemain
- selectedToken:Token
# isSelected:boolean
- game:Game
+ checkSelected():boolean
+ decreaseTokenRemainder():void
Layout
- mainFrame:JFrame
- panelCont:JPanel
- mainPage:MainPage
- iniBoard:InitialBoard
- cLayout: CardLayout
- game:Game
# play:ActionListener
# restart:ActionListener
# close:ActionListener
+ setGame(Game game):void
+ getGame():Game
+ setIniBoard(InitialBoard iniBoard):void
+ getIniBoard():InitialBoard
Board
- intersection:Intersection
- intersectionList:ArrayList
+ getIntersectionList():List
+ getIntersectionPanelIndex(Intersection intersection):int
+ getIntersectionByCoordinate(int x, int y):Intersection
Intersection
- coordinateX:int
- coordinateY:int
- inter:IntersectionPoint
+ Position:enum
+ getCoordinateX():int
+ getCoordinateY():int
+ choosePosition(Position position):void
0..18
has
1
1
has
1
1
has
1
24
1
has
Backend
Application (driver)
+ main(String[] args)
Board
+ MAX_LAYERS:int = 4
+ MAX_POSITIONS:int = 9
- intersection:Intersection[][]
- intersectionList:ArrayList
+ getIntersection(int layer, int position):Intersection
+ makeNewIntersection(int layer, int position):Intersection
+ placeToken(Player player, Intersection intersection):boolean
+ moveToken(Player player, Intersection intersection, Intersection otherIntersection):boolean
Game
+ MAX_TOKENS_PLAYER:int = 9
+ MIN_TOKENS_PLAYER:int = 2
- player1:Player
- player2:Player
- currentPlayer:Player
- board:Board
- isGameOver:boolean
+ endTurn():void
+ swapPlayers():void
+ getBoard():Board
+ getCurrentPlayer():Player
+ isGameOver():boolean
<<abstract>> GameMode
+ startGame():void
+ endGame():void
+ playGame():void
+ pauseGame():void
+ resumeGame():void
+saveGame():void
+ loadGame():void
+ quitGame():void
TutorialMode
NormalMode
Intersection
- layer:int
- position:int
- token:Token = null
+ getLayer():int
+ getPosition():int
+ addToken(Token token):void
+ removeToken():void
+ getToken():Token
+ isAdjacent(Intersection other):boolean
+ isEmpty():boolean
ActionList
- actionSet:Set
+ allowedActions(AllActions actions):boolean
+ addAction(AllActions actions):void
+removeAction(AllActions actions):void
+actionList():List
Player
- name:String
- tokenColour:TokenColour
- tokensOnBoard:int
- tokensInHand:int
- canFly:boolean
- allowableActions:ActionList
+ playTurn():void
+ getName():String
+ getTokensOnBoard():int
+ getTokenColour():TokenColour
+ getTokensInHand():int
+ canPlayerFly():boolean
+ placeTokenOnBoard():void
+ loseTokenOnBoard():void
+ isActionAllowed(AllActions actions):boolean
+ addAllowableAction(AllActions action):void
+ removeAllowableAction(AllActions action):void
+ getAllowableActions():List
Token
- tokenColour:TokenColour
+ getTokenColour():TokenColour
+ setTokenColour(TokenColour tokenColour):void
<<Enum>> AllActions
+ DO_NOTHING
+ CAN_FLY
+ PLACE_TOKEN
+ MOVE_TOKEN
<<Enum>> TokenColour
+ PLAYER_1_WHITE
+ PLAYER_2_BLACK
<<abstract>> Action
+ execute():boolean
+ menuDescription(Player player):String
PlaceTokenAction
- player:Player
- placeIntersection:Intersection
- game:Game
+ execute():boolean
+ menuDescription(Player player)
MoveTokenAction
- player:Player
- currentIntersection:Intersection
- newIntersection:Intersection
- game:Game
+ execute():boolean
+ menuDescription(Player player):String
1
has
1
has
1
1
1
1
has
24
1
has
0..1
1
has
1
has
2
0..18
1
has
0..9
1
has
0..4
1
performs
1
1
has
1
1
1
has

1.2. Design Rationales

- Highlighted in grey: removed from Sprint 1, Highlighted in blue: Updated for Sprint 2

Backend

GameMode is a composition of Game. Abstract class GameMode is a child class of a parent class Game. Every Game of Nine Men's Morris has a GameMode and GameMode will not exist if Game does not exist. GameMode is an abstract class because there will be multiple child classes that inherit abstract class GameMode into specific different game modes.

NormalMode extends GameMode, **TutorialMode extends GameMode.** NormalMode and TutorialMode inherit GameMode abstract class as they are specific game modes for the Game class. This relationship satisfies the **Open-closed principl**e of the SOLID principle as the GameMode parent class is able to be extended into multiple child classes without modification to the parent class. This also satisfies the **Don't Repeat Yourself** (DRY) principle. For instance, if we are adding more modes in the future, we can simply extend them from GameMode, instead of repeating the code that is similar in every class.

Board is a composition of Game. Board is a child class of a parent class Game. Every Game of Nine Men's Morris has a single Board and Board will not exist if Game does not exist. Board is where the game of Nine Men's Morris will be played.

Intersection is a composition of Board. Intersection is a child class of parent class board. The Board in every game of Nine Men's Morris has 24 Intersections and these Intersections cannot exist if there is no Board. Each Intersection acts as a place where the player can place their tokens.

Player is a composition of Game. Player is a child class of a parent class Game. Every Game of Nine Men's Morris has 2 players and players will not exist if there is no game. 2 players play the game of Nine Men's Morris by taking turns to perform actions.

Player---<<has>>--->Token and Game---<<has>>--->Token. Each Player has 9 objects of Token class at the start of the game and it may decrease as the game of Nine Men's Morris goes on. Each Game starts with all its Intersections of the Board as empty and it can be filled with 18 objects of Token class maximum if both Players placed all their Token objects.

Intersection---<<has>>--->Token. Every Intersection on the board is empty when the game of Nine Men's Morris starts and each Intersection can be filled with a single object of Token class.

WhiteToken extends Token, **BlackToken extends Token.** WhiteToken and BlackToken inherit the Token abstract class. WhiteToken and BlackToken inherit all the methods and attributes from the Token abstract class. The only difference between WhiteToken and BlackToken is their colour to differentiate between them as they are held by different Players. This relationship satisfies the **Open-closed principle** of the SOLID principle as the Token parent class is able to be extended into multiple child classes without modification to the parent class. This also satisfies the **Don't Repeat Yourself** (DRY) principle.

* The implementation of BlackToken and WhiteToken is removed due to them being redundant in identifying the ownership of the Tokens as now we can reduce 2 classes that are barely used in the implementation for backend. **Token is now a concrete class in the backend.**

Token---<<has>>--->TokenColour. Token now has an attribute of TokenColour where TokenColour is an enum corresponding to each of the players in the game. BlackToken and WhiteToken are now removed. Token object now uses the private attribute of TokenColour to differentiate the ownership of the Token. This relationship satisfies **Single Responsibility Principle** of SOLID principles as now TokenColour used to differentiate between Token are separated from the Token class.

Token has an aggregation relationship with Mill. A Mill is formed by having 3 objects of Token class from the same owner (e.g. 3 WhiteTokens or 3 BlackTokens) placed in a certain Intersections of the Board. A Mill has to have 3 objects of Token class but objects of Token class can exist without Mill class, thus they have an aggregation relationship.

Mill is a composition of Board. Mill is a child class of a parent class Board. Every Board can have 0 to 16 Mills and the Mills cannot exist if there is no Board as there will be no possible way to form them.

Player---<<has>>--->ActionList. Player now has an Arraylist of AllActions as ActionList. The ActionList stores all of the actions that the Player is allowed to perform.

ActionList---<<has>>--->AllActions. ActionList contains a set of AllActions. This list represents the actions that are allowed to be performed by a certain player. AllActions is stored in a set because the actions stored within it should be unique as it is an identifier if a player can perform certain actions so duplicates are not allowed.

Player---<<performs>>--->Action. Player creates an object of the Action abstract class to perform various actions when it is his turn. Players can use actions to perform actions onto the game such as restarting the game, quitting the game and turning on hints. Players can also use actions to perform moves onto the board and doing this will end the player's turn.

RestartAction extends Action, **QuitAction extends Action, HintAction extends Action, MoveTokenAction extends Action, PlaceTokenAction extends Action.** These 5 types of Actions are concrete classes that extends the abstract class Action, they inherit the attributes and methods of Action but they each have their own specialised methods or attributes to help me affect the Game in a different way, RestartAction will restart the Game into a new Game, QuitAction will stop and exit the current Game, HintAction can be toggled to show the valid moves that the Player can perform, MoveTokenAction will allow the Player to affect the Board, PlaceTokenAction allows the Player to place a token on an empty intersection of the board. This relationship satisfies the **Open-closed principle** of the SOLID principle as the Action parent class is able to be extended into multiple child classes without modification to the parent class. This also satisfies the **Single Responsibility Principle** which defines that a class should only have one job. Instead of having all the methods such as restart, quit, hint and move in the Action class (God class), we create classes for each action ensuring that each class is only responsible for one single part of the application's functionality.

Frontend (every classes mentioned below are added in Sprint 2)

Layout---<<has>>--->[Backend]Game, Layout---<<has>>--->InitialBoard. Layout has a Game object from the backend package as attributes, a setter and a getter method for Game. This is because when the player clicks on the play button, a new Game instance is created in the Layout constructor, to make sure that the Game is always a new Game once the player clicks the play button. Besides, InitialBoard's set game method will also need to be invoked to update the Game instance for InitialBoard. So the relationship between Layout and InitialBoard and Game is association.

InitialBoard---<<has>>--->Board, Board---<<has>>--->Intersection. Board is created by 24 intersections and multiple lines. And, the Board class will be added into InitialBoard, so that Player can visually understand where to place tokens. Besides that, InitialBoard will add Action Listener to the intersections in Board, so that all intersections can be placed by Token and Token can be selected to move.

InitialBoard---<<has>>--->Token, WhiteToken and BlackToken extends Token. InitialBoard has an ArrayList of Token objects to loop through the Token list and check whether the tokens are selected. Thus, they are having an association relationship between each other. Token is an abstract class, WhiteToken and BlackToken extends Token abstract class, as they have the same private attributes as Token class, like coordinate and initial board. With abstraction, we satisfied the **Open-closed Principle** by extending the Token class without modifying the class itself. In the future sprints, if WhiteToken has different attributes from BlackToken, we simply just add in the WhiteToken class.

InitialBoard---<<has>>--->[Backend]Game. InitialBoard has an instance of Game object from the Backend package as attributes. This is because InitialBoard uses the Game attribute to allow invoking several backend objects and methods using the GUI by adding in ActionListeners. InitialBoard needs the corresponding backend Game object to invoke the relevant methods from the backend.

1.3. Discarded Alternatives

Game---<<has>>--->Hint, Game---<<can>>--->Quit, Game---<<can>>--->Restart. We originally designed for Game to have an association relationship between Game with Hint, Quit and Restart for it to have these functions. However, we think that these actions at its core are the same as the MoveTokenAction the Player used to affect the Board. To satisfy the **Don't Repeat Yourself** (DRY) principle, we moved these classes to be a child class of an abstract Action class that covers all the actions that the Player can make.

Game---<<has>>--->GameMode. We originally designed for Game to have a direct association relationship with concrete class GameMode. However, we realised that this approach is not scalable and it violates the **Single Responsibility Principle** of SOLID principles. This is due to GameMode having to manage multiple types of game modes in its class if there is a need to increase the number of game modes. We decided to change GameMode into an abstract class where different game modes can directly inherit its main attributes and methods.

Game---<<has>>--->Player. We originally designed for the Game class to have an association relationship with the Player. However, we think that there is no meaningful way for a Player to exist if Game does not exist. Hence we designed Player as a composition of Game.

Move is a composition of MoveTokenAction. Abstract class Move is a child class of a parent class Move. Every MoveTokenAction has an object of Move abstract class and the Move cannot exist if there is no MoveTokenAction so there is no possible way to move affect the Board without a MoveTokenAction. Move is used to affect the status of the board.

Move---<<is validated on>>--->Board. Any object of Move class will be validated on the Board to ensure it is a valid move based on Nine Men’s Morris rules before it is able to affect the Board..

PlacingMove extends Move, **FlyingMove extends Move, SlidingMove extends Move.** These 3 concrete classes extend the Move abstract class. PlacingMove is used by the Player to place a Token onto an empty intersection on the Board. FlyingMove is used by the Player to move a token to any intersection if and only if they have less than 3 tokens left. SlidingMove is used to move a token to an adjacent intersection. This relationship satisfies the **Open-closed principle** of the SOLID principle as the Move parent class is able to be extended into multiple child classes without modification to the parent class. This also satisfies the **Don’t Repeat Yourself** (DRY) principle.

For the **Move** abstract class that we plan to implement during Sprint 1, we removed them for Sprint 2. After discussion with the team members, we think that PlaceTokenAction and MoveTokenAction are somewhat similar to other Actions, which extend from the Action abstract class, that consists of an execute method and description method. However, we might still change our implementation in future sprints, when more and more features are coming in until we can truly differentiate MoveAction class with other actions extending from Action class. For now, we think that MoveAction class is a bit redundant as it is somewhat similar to Action class, as PlaceToken and MoveToken are also actions performed by the player, which needs the attributes of Action class. This will fulfil the **Don’t Repeat Yourself** (DRY) principle, by not repeating the same attributes and methods in Move class and Action class.

2. APPENDIX

Class Diagram

https://lucid.app/lucidchart/72bd3d5f-67f6-485c-8377-ad7802a7a456/edit?viewport_loc=-422%2C-772%2C4742%2C2345%2C_HFSLbnAqfA8&invitationId=inv_ca1797a5-fe22-4640-8cd0-134bc8e9cfe4