MONASH University

FIT3077
SPRINT 2: INITIAL DEVELOPMENT

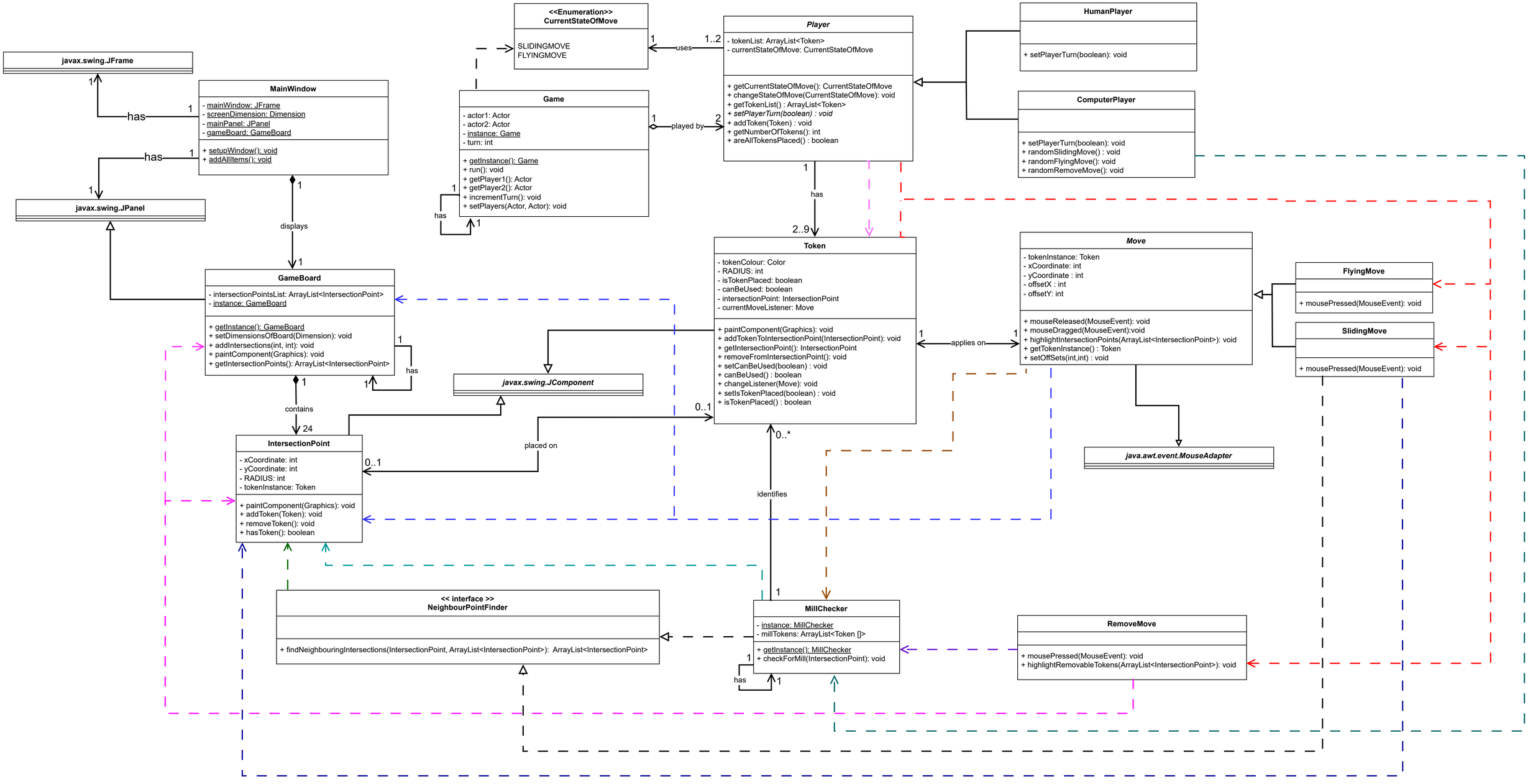
Done By: The Dev Dynasty

Shoumil Guha (32700660)

Rachit Bhatia (32638396)

Tan Jun Yu (32025130)

UML CLASS DIAGRAM


javax.swing.JFrame
MainWindow
- mainWindow: JFrame
- screenDimension: Dimension
- mainPanel: JPanel
- gameBoard: GameBoard
+ setupWindow(): void
+ addAllItems(): void
has
has
displays
javax.swing.JPanel
<<Enumeration>>
CurrentStateOfMove
SLIDINGMOVE
FLYINGMOVE
uses
Player
- tokenList: ArrayList<Token>
- currentStateOfMove: CurrentStateOfMove
+ getCurrentStateOfMove(): CurrentStateOfMove
+ changeStateOfMove(CurrentStateOfMove): void
+ getTokenList() : ArrayList<Token>
+ setPlayerTurn(boolean) : void
+ addToken(Token) : void
+ getNumberOfTokens(): int
+ areAllTokensPlaced() : boolean
HumanPlayer
+ setPlayerTurn(boolean): void
ComputerPlayer
+ setPlayerTurn(boolean): void
+ randomSlidingMove() : void
+ randomFlyingMove(): void
+ randomRemoveMove(): void
Game
- actor1: Actor
- actor2: Actor
- instance: Game
- turn: int
+ getInstance(): Game
+ run(): void
+ getPlayer1(): Actor
+ getPlayer2(): Actor
+ incrementTurn(): void
+ setPlayers(Actor, Actor): void
played by
has
has
GameBoard
- intersectionPointsList: ArrayList<IntersectionPoint>
- instance: GameBoard
+ getInstance(): GameBoard
+ setDimensionsOfBoard(Dimension): void
+ addIntersections(int, int): void
+ paintComponent(Graphics): void
+ getIntersectionPoints(): ArrayList<IntersectionPoint>
has
contains
javax.swing.JComponent
Token
- tokenColour: Color
- RADIUS: int
- isTokenPlaced: boolean
- canBeUsed: boolean
- intersectionPoint: IntersectionPoint
- currentMoveListener: Move
+ paintComponent(Graphics): void
+ addTokenToIntersectionPoint(IntersectionPoint): void
+ getIntersectionPoint(): IntersectionPoint
+ removeFromIntersectionPoint(): void
+ setCanBeUsed(boolean) : void
+ canBeUsed() : boolean
+ changeListener(Move): void
+ setIsTokenPlaced(boolean) : void
+ isTokenPlaced() : boolean
applies on
Move
- tokenInstance: Token
- xCoordinate: int
- yCoordinate : int
- offsetX : int
- offsetY: int
+ mouseReleased(MouseEvent): void
+ mouseDragged(MouseEvent):void
+ highlightIntersectionPoints(ArrayList<IntersectionPoint>): void
+ getTokenInstance() : Token
+ setOffSets(int,int) : void
FlyingMove
+ mousePressed(MouseEvent): void
SlidingMove
+ mousePressed(MouseEvent): void
java.awt.event.MouseAdapter
IntersectionPoint
- xCoordinate: int
- yCoordinate: int
- RADIUS: int
- tokenInstance: Token
+ paintComponent(Graphics): void
+ addToken(Token): void
+ removeToken(): void
+ hasToken(): boolean
placed on
identifies
<< interface >>
NeighbourPointFinder
+ findNeighbouringIntersections(IntersectionPoint, ArrayList<IntersectionPoint>): ArrayList<IntersectionPoint>
MillChecker
- instance: MillChecker
- millTokens: ArrayList<Token []>
+ getInstance(): MillChecker
+ checkForMill(IntersectionPoint): void
has
RemoveMove
+ mousePressed(MouseEvent): void
+ highlightRemovableTokens(ArrayList<IntersectionPoint>): void

DESIGN CHOICES RATIONALE

1. KEY CLASSES

a) Move Abstract Class

Why is the Class required?

The Move class is one of the most important classes in the whole system as it is responsible for every prominent action made in the game. The class consists of multiple methods that are required to control the movement of a token by a player. It also contains additional methods that manage the behaviours of each player action on the token. The methods required for controlling the movement of the token are overridden from the parent class MouseAdapter (present in the Java AWT library), hence it extends from MouseAdapter to define unique implementations of these parent methods. Since, the Move class contains several behaviours, it is not appropriate to solely leave it as a single method. If all of the Move implementation were to be included in a single method, the method would be very long (God Function) which prevents the code from being modular, thus in turn making it harder to extend. Additionally, it can't be made into a method alone because the movement of a token requires overriding the methods of MouseAdapter which can't be done unless there is a child class that extends from MouseAdapter.

Why is the Class abstract?

Initially, the team debated on whether the class should be made concrete or abstract. We considered making it concrete since the two initial child classes, FlyingMove and PlacingMove had the exact same implementations so we considered removing the child classes and solely using the Move class. However, doing so would have restricted the design's extension for flying move and placing move. Hence, we finally decided to merge FlyingMove and PlacingMove into a single class FlyingMove since their functioning is exactly the same and shifted the function for *mousePressed* from Move to FlyingMove. This design adheres to the Single Responsibility Principle since FlyingMove is only concerned with its implementation, while the parent class Move takes care of all the common implementations. The *mousePressed* function is overridden by both of the Move child classes since both of the moves have a unique way of moving the tokens around the board.

b) MillChecker Concrete Class

Why is the Class required?

The MillChecker class defines the behaviour responsible for checking if a mill has been formed. It implements the interface NeighbourPointFinder which consists of a default method that defines the logic for finding the neighbouring intersection point for a particular intersection point on the board. (NeighbourPointFinder has been defined as an interface with a default method because the neighbouring intersection point finder logic needs to be used by multiple classes). The MillChecker uses neighbouring intersection points to check if a mill has been formed. The class can't be designed as just a method because multiple individual classes use the mill checking logic (Move, RemoveMove, and ComputerPlayer). If it were to be a method instead of a class, the method would be repeated in multiple classes, which violates the DRY principle. Hence, designing MillChecker is a better alternative.

MillChecker cannot be designed as an interface also because it requires the ArrayList of tokens array as an attribute (interfaces cannot have attributes) which is used to keep track of the tokens that are part of a mill. This ArrayList is used to store the array of tokens that are currently a part of the mill. Once a token is moved out of a mill, the particular array of tokens is removed from the ArrayList.

Why is the class concrete?
The class is concrete because in order for the *checkForMill* method to be accessed by multiple classes, the MillChecker object needs to be instantiated. However, since the object does serve any functionality on its own and it can be reused, we have implemented the Singleton design pattern to avoid unnecessary creation of multiple instances of MillChecker, thus saving memory space.

2. KEY RELATIONSHIPS

a) Interface Implementation relationships of NeighbourPointFinder

The MillChecker and SlidingMove classes implement the NeighbourPointFinder interface because both of these classes use the default method *findNeighbouringIntersections*. This method uses the given intersection point and list of all intersection points on the board to find the neighbouring intersection points. It returns an ArrayList of the neighbouring intersection points. The relationships to this interface are prominent as the implementation of its method defines the subsequent behaviour of the particular classes. MillChecker needs to find the neighbouring intersection points to check if the adjacent neighbours of the given point also have tokens, thus enabling it to check for a mill. SlidingMove uses the neighbouring points to inform the player of the available adjacent points they can move their token to. Since the NeighbourPointFinder uses intersection points in its logic, there is a dependency to the IntersectionPoint class. An association to NeighbourPointFinder has not been used since only the function is used and if it were a class, the class on its own would not serve any purpose. Additionally, the method has been designed as a default method inside an interface to enable easy reusability of code. Thus, the relationships to NeighbourPointFinder are of interface implementation type.

b) Bi-directional relationship between Token and IntersectionPoint

A bidirectional association relationship is used between the Token and IntersectionPoint class since the Token has an attribute of IntersectionPoint while the IntersectionPoint also has an attribute of Token. The main reason for using a bidirectional relationship is to be able to track or reference back to the previous IntersectionPoint that a particular Token was in so that it can be removed from the old IntersectionPoint whenever this Token is moved to a new IntersectionPoint. In other words, if only a single direction association was used from the IntersectionPoint to Token, it is feasible to add this Token to the new IntersectionPoint but it can be difficult to track back the IntersectionPoint that the Token was previously in to remove this Token instance from it. Hence, whenever a Token is added to an IntersectionPoint, the Token needs to be added to the attribute of IntersectionPoint where it belongs to and the corresponding IntersectionPoint also needs to be added to the attribute of Token.

3. INHERITANCE DECISIONS

a) Inheritance between Player with HumanPlayer and ComputerPlayer

The Player class is inherited by both HumanPlayer and ComputerPlayer because it is known that these two child classes have common attributes and methods with one another besides having some methods of their own. For instance, both the child classes will have an ArrayList of Tokens and CurrentStateOfMove as attributes alongside the methods of getters and setters. This allows the same code to be reused without having to violate the Don't Repeat Yourself(DRY) principle. Besides that, Player class is declared as an abstract class since it will not be instantiated and most importantly the child classes which are HumanPlayer and ComputerPlayer have a common method that is the setPlayerTurn() but with different operations or implementations and therefore requires the method to be abstract. The setPlayerTurn() of the HumanPlayer is allowing the tokens to be moved by dragging them while the setPlayerTurn() of ComputerPlayer will be used to call the generated random move methods in its own class.

b) Inheritance between Move with FlyingMove and SlidingMove

Move class is inherited by both FlyingMove and SlidingMove because it is known that these two child classes have common attributes and methods with one another. For instance, both of the child classes will have Token, xCoordinate,yCoordinate,offSetX and offSetY as attributes alongside the mouseEvents methods. Both the mouseDragged and mouseReleased will have the exact same implementation between FlyingMove and SlidingMove and therefore they are placed in the parent class Move itself. This allows the same code to be reused without having to violate the Don't Repeat Yourself(DRY) principle. Only mousePressed has different implementations and thus it is placed in the child classes themselves.

An alternative to using an inheritance would be having an if-else statement differentiating the implementations of the mousePressed method depending on Flying or Sliding without using an inheritance since mousePressed is the only method with different implementations.In that case, FlyingMove and SlidingMove will not be classes on their own because these two child classes do not have any other attributes or methods of their own.However, considering the extensibility of the code, it is decided that FlyingMove and SlidingMove should be created as separate classes inheriting the Move abstract class so that it allows new features or implementations to be added with ease in the future.

4. KEY CARDINALITIES

a) Cardinalities between Player and CurrentStateOfMove

Both players can only have a single CurrentStateOfMove at any given time during the duration of the game. If the player has 3 tokens left, they can move the token to any board, so the current state of the move the player can make has to be that it can only fly (hence FlyingMove).

Likewise, if the player has more than three tokens on the board, they are only allowed to move the token to an adjacent intersection point. Therefore, the current state of the player's moves only allows it to be slid (hence SlidingMove).

The choice to use enums was so that the game can only make decisions on how the player moves based on two specific states. It removes ambiguity from the way the game operates and how each player interacts with it.

b) Cardinalities between MillChecker and Token

For our design. It was decided that the game shall only use one instance of MillChecker to check for mills present for a token in the game.

During the game, the same MillChecker can be called on the token to check whether it forms a mill when it is moved to an empty intersection point. It can be used for all the tokens on the board simply by passing in whichever token you wish to check a mill for. Hence, there is only one MillChecker for every token on the board.

It should be noted that the MillChecker is a singleton class (i.e. A class that instantiates one object only) since multiple instances are not necessary. This is elaborated on further in the next section.

5. USAGE OF DESIGN PATTERN

The Singleton Design Pattern has been used in our software architecture. Three classes implement the singleton design pattern - Game, GameBoard, and MillChecker. The Singleton pattern has been used either to prevent unnecessary creation of multiple instances, or to ensure the same instance is used throughout the existence of the application. MillChecker has a Singleton instance to avoid unnecessary creation of multiple instances because only the method of the class needs to be used, while the MillChecker object on its own does not serve any major purpose. On the other hand, the Game class and the GameBoard class have singleton instances because these classes form the core of the application, hence, multiple instances should not be created for them. For example, the Game class manages the run of the game, while the GameBoard forms the main board display. Thus, there is no need for these classes to have multiple instances because the same object will be managing the game throughout a single run of the game. These classes were also not made abstract because they have multiple methods and attributes, so all of the methods and attributes would have to be made static which makes the code harder to maintain. The Singleton pattern fits very well in these cases and helps enable global access to these instances, hence, the Singleton pattern has been used in our architecture.

Discarded Alternatives:
A feasible alternative which could be applied to our architecture could be the Factory Method pattern. However, the factory method is required when there are different types of instances that need to be created. In our design, the Game, GameBoard, and MillChecker classes do not require instantiation of different types of instances under a single factory method as these classes are singular and are not extended. Hence, the factory method pattern would just create additional complexity and the multiple types of instances would never be used leading to redundancy.

Another alternative that was considered was the Monostate design pattern. It allows for multiple instances to be created while keeping the values of all the instances the same. While this design pattern fits in our architecture, creation of multiple instances is still never required by the system. The monostate design may provide more transparency, but the system can be easily handled just by using the *getInstance* method of the singleton class. Also, it is better to use the same instance of Game and GameBoard for greater consistency. Hence, the singleton design is a better option as it saves memory space and keeps the usage of these classes straightforward.

REFERENCES

The team acknowledges the use of ChatGPT (https://chat.openai.com/) to learn more about the available design patterns. The following prompts were entered on 22 April 2023:

- “Give some design patterns with explanation”
- “Give design patterns similar to the singleton pattern with explanation”

The output from the generative AI was used as a source to gain insights on the available design patterns. Based on the knowledge gained, a discussion was conducted among the team to choose the best design pattern to implement in the software architecture among all the feasible options.