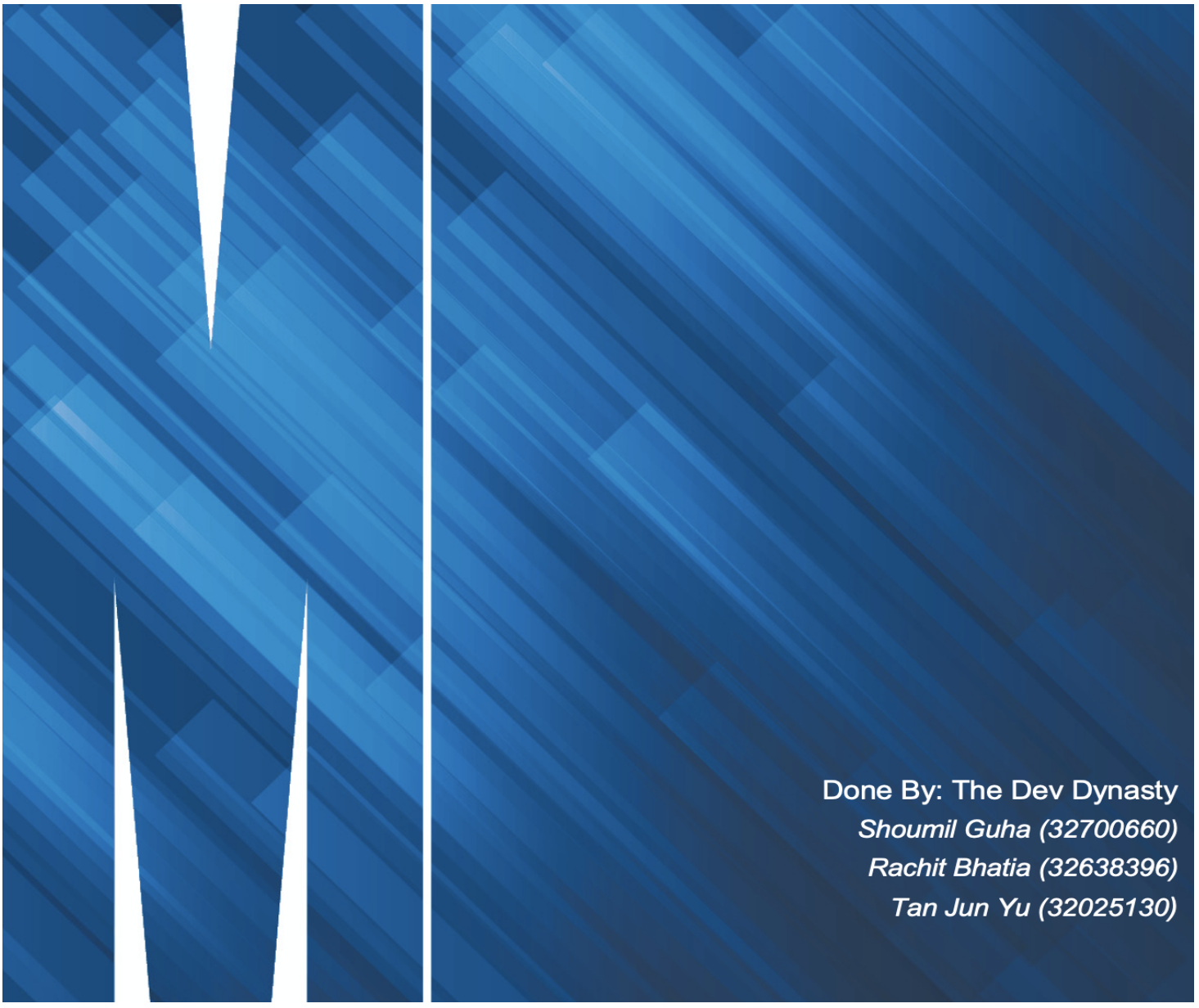
MONASH University

FIT3077
SPRINT 3: PRODUCT DEVELOPMENT



Done By: The Dev Dynasty

Shoumil Guha (32700660)

Rachit Bhatia (32638396)

Tan Jun Yu (32025130)

ARCHITECTURE & DESIGN RATIONALE

1. What has changed and Why has it changed?

Type of Change	Changes	Justification
New Class	PerformThread class added	This class is added to create a new background dispatch thread which handles all the Game loop operations. A new thread separated from the main thread is created so that the UI updates of the app can be done without lag and without overloading the thread. The run method handles the Game loop.
New Class	MainPagePanel class added	This is the class responsible for the main home page of the game and contains the button which starts the Game loop. The function createPlayButton creates the main play button. The class sets the main window displays for the home page and the game page, and sets the players for the game. MainPagePanel also creates an instance of PerformThread which is used to set the Game loop running.
New Relationship	Implementation relationship from Game to NeighbourPointFinder	The Game class implements the NeighbourPointFinder interface because it uses the neighbour finder method to check if a player has any valid move left by checking if any token of that player can be moved to an adjacent available position. This check is done in the checkIfPlayerHasNoSlidingMove method.
Change of class to Singleton type	The MainWindow class was changed to a Singleton class	The MainWindow class had all static attributes and methods initially. However, the need for additional attributes and methods grew with the application (like playerLabel and playerStateOfMoveLabel), hence instead of having too many static instances, the class was converted to Singleton since it is a trusted design pattern. All the attributes need to belong to the same instance because they are supposed to remain constant for the entire lifetime of a game object since they represent the states of the game. Since it was made to a singleton class, a new self-association is introduced because it contains an instance of itself (singleton static instance).
New attributes	playerLabel and playerStateOfMoveLabel attributes added along with their setters and getters added to MainWindow class	These represent the player labels displayed on the screen. They are set as attributes because the state for these attributes is controlled in the Game class. For example, the playerLabel attributes are used to draw the border displaying player turn in the Game loop.
New Enum Value	A new element named PLACING is added to the CurrentStateOfMove Enumeration Class	It was initially thought that PLACING is not needed in the enumeration as the implementation of placing a token is no different than flying. However, PLACING is now added to the enumeration class as we still need to differentiate between PLACING and FLYING since

		placing a token is only done once for each token and we will need to remove the mouseListener from it after it is placed. Hence, this PLACING is used in a if statement to check if the player's currentStateOfMove is PLACING, and if it was PLACING then we will remove the mouseListener. Other than that , PLACING is needed for the switching of state of move to SLIDING and thus it is also used in an if statement to check if current state of move is PLACING with extra condition whether or not all the tokens are already placed before switching to SLIDING.
Replacement of method	changeStateOfMove(Move) is removed and replaced with updateStateOfMove()	The main reason for this change is to avoid violating the Don't Repeat Yourself(DRY) principle. As can be seen from the signature of the changeStateOfMove(Move) method , a designated Move has to be passed in as a parameter. With that being so , the new Move must already been known and therefore in the game loop there are two repeated blocks of code one for each player with a few if statements to check the current state of the game and switch to new Move if needed. Since both the players have the same implementation of checking the state of game and changing state of Move, they are now combined into a single method instead and this reduces a lot of repeated code in the game loop. In the game loop , we now only require to call player1.updateStateOfMove() and player2.updateStateOfMove().
New attribute	A new boolean isMoveValid is added to Intersection Point Class with its setter and getter	isMoveValid is mainly to differentiate the Intersection Points that are valid to be placed with a particular Token and those that are invalid. E.g. If the current state of move of a player is SLIDING, when a token is picked up , only the neighbouring adjacent positions are valid intersection points and the rest are invalid.
New attribute	A new boolean pointSelected is added to Intersection Point Class with its setter and getter	pointSelected is used to check whether a particular token has been pressed to be moved on the board and indicates the availability of a position on the board. This boolean value is used in the paintComponent() method of IntersectionPoint. If set to true, a green border is painted on the intersection point indicating it is an available position on the board for placing the token.
New method	resetAllIntersectionPoints() is added to the GameBoard Class	This method is added to GameBoard as this Class has all the 24 intersection points . Whenever a token is selected or picked up by a player, only the Intersection Points that are valid can be placed with that token. In order to differentiate between valid and invalid intersection points to place a token , setMoveValid(true) is used on the intersection points that are valid positions. The reason to add this method resetAllIntersectionPoints() is to reset all the intersection points back to invalid positions by calling the method setMoveValid(false) on all intersection points.

		Hence, the method resetAllIntersectionPoints is needed whenever a different Token is selected because different tokens can have different Intersection Points that they are allowed to place on.
Modification of method signature	drawLines(int,int,int,int,int, Graphics) in GameBoard class	To draw lines between the intersection points on the game board. The implementation for this was initially very long with multiple for loops and by encapsulating it as a method now reduced a lot of repeated code which is not needed that could just be passed in as parameters instead.
New method	checkIfPlayerHasValidSliding Move() added to the Game class and the Game class now implements NeighbourPositionFinder interface	This method is added to the Game Class as this Class has the game loop and the method is needed to ensure both players have valid sliding positions to keep the game running. If it happens that either of the players no longer has a valid sliding move for all of its tokens, the opponent should win and the game loop terminates. In order to check if there are sliding moves available, NeighbourPositionFinder is needed to see if there is at least one empty adjacent position for at least one token
New attribute	player (:Player) attribute is added to the Token Class with its relevant getter and setter.	It is needed to know if a Token belongs to which Player when checking if a mill is formed . This is mainly to differentiate the tokens of Player 1 and Player 2 during a mill check as the 3 tokens aligned either vertically or horizontally may not belong to the same player. A mill is formed only if all the tokens aligned are of the same Player.
New method	removeTemporaryListener() is added to the Token Class	This method is needed mainly for the removal of a token after a mill is formed. When a mill is formed by a player , the opponent's tokens that are not part of a mill will be set to a RemoveMove listener so that one can be selected to be removed from the game. However, this RemoveMove listener is only needed for a temporary period until a token is selected by the player to remove and the tokens need to be reset back to their original mouseListener before they were set to RemoveMove. The method removeTemporaryListener is used to remove the RemoveMove listener from the tokens and reset it back to the original mouseListener.
New Attributes	toRemove and inMill boolean attributes added to Token class	Both of these boolean values are used to control the UI state for tokens. They are used in the paintComponent method to check for the kind of highlight required for tokens. These are set as attributes so that the value can be controlled from other classes and are set to public visibility because there is no additional guardian code required for setting these values, hence setters are not used. toRemove is used to highlight the token red when it is in removable state, and inMill is used to highlight the token blue when it is part of a mill.
New method	checkHorizontalVertical() is	This method is to avoid code smell of having a long

	added to the MillChecker Class	method. This method is extracted out from the checkMill() method which was initially containing too many lines of code. It is used specifically to look for horizontal mill or vertical mill after a token is placed on an intersection point.
Replacement of method	highlightRemovableTokens() method was removed from the RemoveMove class and replaced with changeToRemoveState(Player) in MillChecker class	The initial design was to have a highlight method in RemoveMove which would highlight all removable tokens of the opponent player in red. However, it was a better option to shift this method to the MillChecker class since it has the attribute that stores all mills which is needed to confirm the tokens to highlight. If this method was kept in RemoveMove as originally planned, the attribute MillTokens would have to be passed in as a parameter for RemoveMove which would increase coupling between the 2 classes.
New Method	checkIfTokenInMill(Token) is added to the MillChecker class	This method is needed to check if a token which was previously a part of the mill has been moved so that its mill record could be removed from the system if it has moved out of the mill. Initially it was planned to keep this as a part of the checkMill function but this violates Single Responsibility Principle and makes the code harder to maintain. Furthermore, this function is also required by RemoveMove, hence designing it as a separate method helps in code reusability without redundancy.
New Attribute	tokenInstance (:Token) added to RemoveMove	The token attribute is needed in RemoveMove to be able to set the opponent player's tokens' state to removable. It is used to identify the particular token the remove click has been performed on, thus removing it from the game.

2. Non-Functional Requirements

Usability

The design of the 9MM application has taken into account Usability as a non functional requirement. The user interface is made simple and easy to navigate even for a new user. Besides that , the display of the application fits for all sizes of different devices as the team has employed a responsive design that will capture and adapt to any screen dimensions of the user. This eliminates the effort of user resizing the application whenever it is launched. The highlighting feature of the application also helps the user to identify the events that happen in the game . For instance, the valid intersection points to place a token are highlighted in green, a mill formed is highlighted in blue and also any removable token is highlighted in red. It is also easy to determine which player's turn it is in any point of the game as the player label will be highlighted in green where it is his or her turn. Moreover, the current state of move of each player is displayed at all times to ease the user in identifying what are the valid moves(PLACING, SLIDING, FLYING) that they can make during their turns.

Maintainability

The software architecture of the code base for this application is modular and decoupled. Object oriented programming is strictly applied during the code development process that allows components to be well separated in doing their own respective responsibilities . This ensures minimum dependencies and coupling between classes which ease the process of maintenance as any modifications needed in the future may not as heavily impact the current source code which might end up affecting overall game behaviours. Other than that, the code is well documented with detailed explanations leaving no ambiguities for the readers. For instance, in-line comments are added for complex code blocks; also each function has a function header documentation that briefly describes the purpose of the function. Moreover ,complex if-else statements with too many lines of code are avoided by trying to reduce them into a much simpler way without affecting their functionalities.

Extensibility

The implemented design is easily extensible because of the abstractions and modularities present in the design. Abstractions such as the interface NeighbourPointFinder and abstract classes Player and Move enable easy modifications and applicability of code. The abstract classes provide an easy way to implement functions for child classes in specific ways based on their personal requirements just by overriding the methods. This feature has been used in our implementation at several instances, for example, the mousePressed function is overridden by the child classes SlidingMove and FlyingMove and each of them have their personal implementations based on the requirements. The design can be further extended by overriding the methods and creating requirement-specific implementations. The NeighbourPointFinder interface promotes easy extension by reducing coupling between classes and enabling dependency of higher level modules on interfaces rather than lower level modules. The Game, MillChecker, and SlidingMove classes implement this interface and hence, have no direct dependencies between each other. Extensibility is important for the 9 Men’s Morris game because it is possible that new rules or new game modes may be introduced into the game in the future to keep the game more interesting. For example, to accommodate new game modes, extensibility is a very important criterion so that any new feature can be easily implemented without major changes to the game mode. Since new rules and modes are very common for any game, extensibility is an extremely important NFR for this project.