Rationale

1. What has changed and WHY has it changed? (JUN YU & RACHIT)

Changes	Justification
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.
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 reduced alot of repeated code in the game loop. In the game loop , we now only require to call player1.updateStateOfMove() and player2.updateStateOfMove().
A new attribute Player 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.
A new attribute isMoveValid is added 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.
A 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.
A new method checkIfPlayerHasValidSlidingMove() 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 atleast one empty adjacent position for atleast one token
A new method removeTemporaryListener() is added to the Token Class	This method is needed mainly for the removal of 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 original mouseListener.
A new method checkHorizontalVertical(IntersectionPoint,ArrayList<Token>,boolean) Is added to the MillChecker Class	This method is to avoid code smell of having a long 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.
drawLines(int,int,int,int,int,Graphis) 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.

2. 2-3 NFR: Why is each relevant (what is its importance?) and how does it fit into our design? How is this shown in our design?

Usability (JUN_YU)

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 the 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 - good documentation / Simple code (no complex if else statements? - easy to read) (SHOUMIL)

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 (RACHIT)

3. Explain one human value that we explicitly considered. Why is it important to our game? How is this shown in our design? (SHOUMIL)

Pleasure (simple UI)