Design principle, purpose, and responsibility of each class and relationship

1. Application

   The Application class is the entry point of the Nine Men Morris game. Its main method is used to instantiate the GameController class to start the game. By having a separate class responsible for the application entry point, we are able to keep the main method clean and simple, which allows us to adhere to the ***Single Responsibility Principle*** by separating the concerns of starting the game from the game logic itself.

2. GameController

   The GameController class takes care of managing the Board and View, as well as handling user clicks and providing information on the current game status. This means that the GameController class holds the instance of both Board and View. As a result, there is a ***composition*** relationship between GameController and Board, and GameController and View, as both classes rely on GameController and cannot function independently. Besides that, this class makes use of the ***Singleton*** design to guarantee that only one GameController exists throughout the entire game. This pattern guarantees that the class behaves consistently throughout the game and avoids needless duplication of the GameController.

   The GameController class includes several methods that manage the game's behaviour.

   - The displayMessage() method displays a message reflecting the current game status.
   - The swapTurn() method switches to another player to perform an action, depending on the current player.
   - The canStartMove() method returns true if all 18 tokens have been placed on the Board, indicating that players can start moving tokens; otherwise, it returns false.
   - The canStartJump() method returns true if a particular player has only 3 tokens left, indicating they can start jumping; otherwise, it returns false.
   - The endGame() method returns true if a particular player has less than 3 tokens or has no possible legal move.

- The runGame() method is responsible for running the game loop, processing user clicks, updating the game's current status, and reflecting changes on the Board and player status.

Additional methods may be added in the future during implementation.
The ***Command-Query Separation Principle*** was followed in the development of the GameController class. It encourages better code readability and maintainability that every method is created to be either a command or a query, but not both. The GameController class keeps a distinct separation between its state-modifying behaviour and its information retrieval behaviour by abiding by this principle.

3. Board

As the foundation of the game's model, the Board class is essential in initialising the Token, and Position instances. It is in charge of carrying out a number of operations, including determining whether any player has created a mill, adding tokens to the board's positions, and controlling the movement of tokens from one position to another. This class contains all the logic needed to manage various board situations and makes sure the game runs smoothly.

A ***composition*** relationship exists between Board and Position, as Position does not make sense to exist independently without Board. The two classes are mutually dependent, and neither can function without the other. In contrast, the relationship between Board and the Token is one of ***aggregation*** rather than composition. This is because Board and Token are independent classes that can exist individually without affecting one another. In other words, ending one class will not result in the termination of the other class.

The Board class is similar to the GameController class in that only one instance is needed to play the full game. To prevent the creation of multiple instances of the Board class, the ***Singleton*** design pattern can be used. This approach not only saves memory resources but also maintains a consistent game state by providing a single source of truth for the board model.

The Board class, which initialises Token, and Position instances, is a crucial part of our game. Its main responsibility is to control how the game board behaves and monitor the game's development. To achieve this, several methods will be included in the Board class.

- The createToken() method is used to instantiate Token instances that will be used in the game. This method ensures that all the required tokens are available for gameplay.

- The createPosition() method is used to create Position instances that define the board's layout. This method ensures that all positions on the board are appropriately defined.
- The isMillFormed() method is responsible for detecting whether a mill is formed on the board by any player. This method returns true if a mill is formed; otherwise, it returns false.
- The moveToken() method is used to move a Token instance from one position to another on the board. This method ensures that players can move their tokens around the board as needed.
- The jumpToken() method is used to change a Token instance's position from one to another. This method ensures that players can move their tokens to any position on the board.
- The removeToken() method is used to remove a Token instance from the board. This method is crucial when a player has used up all their tokens or when a token is captured by an opponent.

The Board class also has a number of getters and setters for retrieving instances of Token, and Position as needed. These techniques make sure that the game's status can be accessed and changed as required. The Board class might get more methods later on during implementation to increase its flexibility and usefulness.

The Board class was developed using the ***Command-Query Separation Principle***. Every method is designed to be either a command or a query, but not both, which promotes improved code readability and maintainability. By adhering to this concept, the Board class maintains a clear distinction between its state-modifying behaviour and its information retrieval behaviour.

4. Position

The Position class is responsible for representing the point of intersection between lines on the board. It stores both the x and y coordinates of its location on the board, as well as the current token placed on it and its neighbouring positions. This information is crucial for the Board class and GameController class to facilitate gameplay.

The Position class follows the ***Single Responsibility principle***, which states that it only concentrates on clearly and succinctly providing its information, without any other responsibilities. Due to how easy it is to evaluate and modify the Position class using this method, code modularity and maintainability are supported.

5. Token

   The Token class is responsible for representing the game pieces used in the game. Players keep track of their number of tokens based on the number of Token instances they possess, while instances of this class are stored in the board to facilitate game logic. The Position class stores instances of Token, and the Token class is responsible for informing the Board or GameController which player the current token belongs to based on its colour. The Token class exemplifies the ***Single Responsibility Principle*** as it has only one role in the game. This strategy minimises the likelihood of code errors and complications while ensuring that the class is focused on its main duty.

6. TokenColor

   The TokenColor enum class serves as a practical way to represent the colour of tokens in the game, with available options of BLACK and WHITE. By defining these states as constants within an enum class, it ***avoids the excessive use of literals*** throughout the code, which can improve readability and maintainability. This approach also makes it simpler for other developers to comprehend the code. The TokenColor enum is linked to the Token class and utilised by the Board and GameController classes to streamline the game logic.

7. View

   The abstract View class is used to provide a high-level abstraction of the game's user interface (UI) by encapsulating the common behaviour and properties of these UI components such as methods for rendering and updating their content. In a way, this abstract View class ***reduces code duplication*** across different UI components and promotes code reuse. At the same time, the abstract View class also follows the ***Dependency Inversion Principle*** by decoupling the higher-level modules such as the GameController class from the lower-level modules such as the BoardView, MessageView and InitialPageView class. This helps to promote flexibility in the system and enables easy swapping of different implementations of View components, without impacting the GameController class.

8. BoardView

   The BoardView class extends the abstract View class to provide a concrete implementation of the UI component responsible for rendering the game board to the user. Apart from inheriting the common methods from the abstract View class for rendering the UI, it also adds specific behaviour and properties required for rendering the game board, such as the position of the

board and the size of each cell for placing the token. Overall, the BoardView class demonstrates the use of ***Single Responsibility Principle*** by having only 1 role, which is to show the game board to the user and handle user interactions related to the board.

9. MessageView

   The MessageView class is a concrete implementation of the abstract View class that provides a UI component for showing messages to the user. This includes notifying the user when it's their turn, whether any mills have been formed, and other relevant game information. By extending the abstract View class, the MessageView class inherits the common methods required for rendering and updating the UI component, while also adding its specific properties such as the font size and colour of the displayed messages. Overall, the MessageView class adheres to the ***Single Responsibility Principle*** by only focusing on presenting a clear and informative interface for the user, ensuring that they are kept informed of the current game state at all times.

10. InitialPageView

    The InitialPageView class extends the abstract View class to provide a concrete implementation of the UI component responsible for displaying the game's main menu to the user. This menu includes buttons for playing against the computer, playing against a human player as well as accessing a tutorial mode guiding new players on how to play the game. One key design principle that the InitialPageView class follows is the ***Single Responsibility Principle*** as it focuses solely on presenting the menu in a clear and concise manner, without being encumbered by other responsibilities. This helps to promote code modularity and maintainability, making it easier to test and modify the InitialPageView class as needed.

11. GameState

    The GameState enum class provides a convenient way of representing different states of the game, such as SET, MOVE, JUMP as well as REMOVE. By defining these states as constants in an enum class, it ***avoids the excessive use of literals*** throughout the code, which can certainly enhance the readability and maintainability of the code, making it easier for other developers to understand. The GameState enum is used by the GameController class to keep track of the current game state, allowing the game to respond accordingly to the user's actions. For example, if the game is in the SET state, the GameController class will handle the player's move as a placement of a new token, while if the game is in the MOVE state, it will

handle the player's move as a movement of an existing token to a new position on the board.

12. GameMode

As for the advanced requirement, we are to create a tutorial mode that guides the player through the gameplay and game rules. Our idea of tutorial mode is to have a predefined position of the game and a series of predetermined steps that the player needs to make. Through the process, the player will learn about move, remove, jump, winning conditions and so on. Having different modes, a GameMode abstract class is constructed. It is the abstraction of different modes and it applies the ***Open Closed Principle (OCP)***. Game mode is opened for extension and closed for modification. We can easily add different game modes without affecting the existing codes. This also with ***Don't Repeat Yourself (DRY)*** where the methods shared by different game modes can be defined at first together and overridden later for more details. It has an association relationship with gameController. GameController requires this attribute to decide what GameMode is delivered.

Our team plans to use ***State Design Pattern*** for the GameMode. It is a behavioural design pattern that alters an object's behaviour when its internal state changes. In our case, we want the GameController to set up a game accordingly based on the game mode. In different states, it will have different behaviours, which is either preparing a normal game or a tutorial game. State-specific methods are constructed to initialise the board and actions. This design pattern applies ***Dependency Inversion Principle (DIP)***. High-level module, GameMode, does not depend on low-level module, NormalMode or TutorialMode. Instead of checking if the game mode is normal or tutorial and execute different implementation for setting up the game, we would just use abstraction. GameController will have a GameMode attribute, which acts as our "state" in the design pattern. GameController would have a method to set up the board, which calls the abstract method of the GameMode class to set up the board. The actual details of implementation would depend on the method that overrides the abstract method.

13. NormalMode

The NormalMode class extends the abstract GameMode class. It refers to the normal mode where there are 2 players, who can be computer or human players, play against each other under the standard rule. This class achieves ***Single Responsibility Principle (SRP)*** as it only serves for the purpose of starting a new game with an empty board.

14. TutorialMode

    The tutorialMode class extends the abstract GameMode class. This mode enables the initialization of predetermined board positions and a series of moves that the player has to follow. Similar to normal mode, this mode obeys ***Single Responsibility Principle (SRP)*** because it only serves for tutorial mode and does not allow normal gameplay. Player is allowed to move as planned and cannot make any other moves. With that said, the tutorial mode can be taken as an interactive demonstration for the player.

15. Player

    An abstract Player class is made. A game has 2 players and we can have different types of players, human or computer. In this case, both humans and computers can perform actions such as move, jump, remove, and set to be able to play the game. Each player also has 2 to 9 tokens depending on the game situation. This justifies that Player class will have association with abstract Action class and Token class. Despite the type of the player, we can observe that the basic mechanism for each player is the same. This leads to the ***Don't Repeat Yourself Principle (DRY)***. The attributes which indicate that the player can perform actions and own tokens are consistent. The difference is mainly on how humans or computers make a move. ***Single Responsibility Principle (SRP)*** is applied here where the Player class is only able to do what a player can do. The player will have tokens and can perform actions to play the game. No extra responsibilities are given to the Player class.

16. HumanPlayer

    HumanPlayer class extends Player class. It overrides the methods from the Player class and defines its implementation of how a player would react to a player's input and perform action accordingly.

17. ComputerPlayer

    ComputerPlayer class extends Player class. It overrides the methods from the Player class and defines its implementation of how a computer would select and perform actions based on a predefined algorithm, which is randomly selecting available moves in our case.

18. Action

    Action class is an abstract class that is executed on the Position class and Token class which provides a high-level abstraction of the actions in the

game. By having this abstract class, all the common properties and behaviors of all the actions such as the methods for taking the token can be inherited by its child directly. Additionally, the abstract Action class reduces the duplication code across different actions in this game and allows all the other actions to reuse the code in this class, which obeys the ***Don't Repeat Yourselves Principle***, which makes our code good maintenance. If more actions need to be done in this game in the future, this follows the ***Open Close Principle*** because when more actions are added, you do not have to modify the Action class but allow the additional class to extend it, this helps to promote flexibility in the system.

19. RemoveAction

    RemoveAction class extends the abstract Action class. It overrides the methods from the Action class to allow one player to remove the opponent's token when the player has a mill. This obeys the ***Single Responsibility Principle*** as it only serves the purpose of removing the token from the board.

20. SetAction

    SetAction class extends the abstract Action class. It overrides the methods from the Action class to allow the player to set the tokens on the board. This obeys the ***Single Responsibility Principle*** as it only serves the purpose of setting the token on the board.

21. MoveAction

    MoveAction class extends the abstract Action class. It overrides the methods from the Action class to allow the player to move the tokens on the board, and it obeys the ***Single Responsibility Principle*** as it only serves the purpose of moving the token on the board from one position to an adjacent position.

22. JumpAction

    JumpAction class extends the abstract Action class. It overrides the methods from the Action class to allow the player to jump the tokens on the board, and it obeys the ***Single Responsibility Principle*** as it only serves the purpose of jumping the token on the board from one position to another position.

Discarded Alternatives

1. Mill

   Initially, a separate Mill class was included in the domain model with an association relationship from the Board class to check if any mill was being formed by either player. However, this led to redundancy in the code as the Board class already had information on all the tokens placed on the board at any given moment. To avoid redundancy and follow the ***Don't Repeat Yourself (DRY)*** principle, the mill checking logic was moved to the Board class itself. By consolidating this logic into the Board class, it helps to not only eliminate the need for a separate Mill class but also ensures that the code becomes more organised and easier to maintain in the long run.

2. Strategy

   To introduce the Strategy design pattern, we initially planned to create an interface named Strategy and implement it with two classes - HumanStrategy and ComputerStrategy. We reasoned that the algorithm to compute moves for the human and computer players was different. Our goal was to use the design pattern to separate the implementation details of the algorithm from the code that calls it and follow the Open-closed principle, allowing for the introduction of new strategies without modifying existing code. However, upon further consideration, we found that this step was unnecessary. We could include these logics within the HumanPlayer and ComputerPlayer classes themselves, eliminating the need for the design pattern in our design.

3. Hint

   Initially, we planned to implement a Hint class that would detect the available legal moves for the current player, whether human or computer, by containing an algorithm for this purpose. However, upon further consideration, we discovered that the algorithm could be incorporated directly into the Player abstract class. This would enable the HumanPlayer and ComputerPlayer classes to inherit and use the algorithm without the need for a separate Hint class. Therefore, we ultimately decided that the Hint class was unnecessary and omitted it from our design.

4. GameRule

   Initially, we had planned to implement a GameRule class to handle various game rules such as checking if a mill is formed, if any player has no legal moves, if a player has less than 3 tokens, and keeping track of the game and

player status. However, upon further consideration, we realised that these checks would be better suited in the GameController and Board classes. Therefore, the implementation of GameRule was deemed unnecessary and was ultimately removed from our design.

Basic UI Design

- Draw low-fidelity (low-fi) prototype drawings of the proposed user interface, cover both the basic 9MM gameplay and the chosen advanced requirements

Initially, we implemented a basic user interface design for our project. However, as we progressed, we realized that the design lacked creativity and did not have the visual appeal that we were aiming for. Therefore, we made a conscious decision to improve upon the existing design and make it more engaging and creative. We wanted to ensure that our users would have a memorable and enjoyable experience while using our application, and a creative user interface was crucial for achieving that. With this in mind, we incorporated more design elements, color schemes, and interactive features to make the user interface more visually appealing and engaging.