Nine Men's Morris Sprint 2

Team Name:

Brute Force

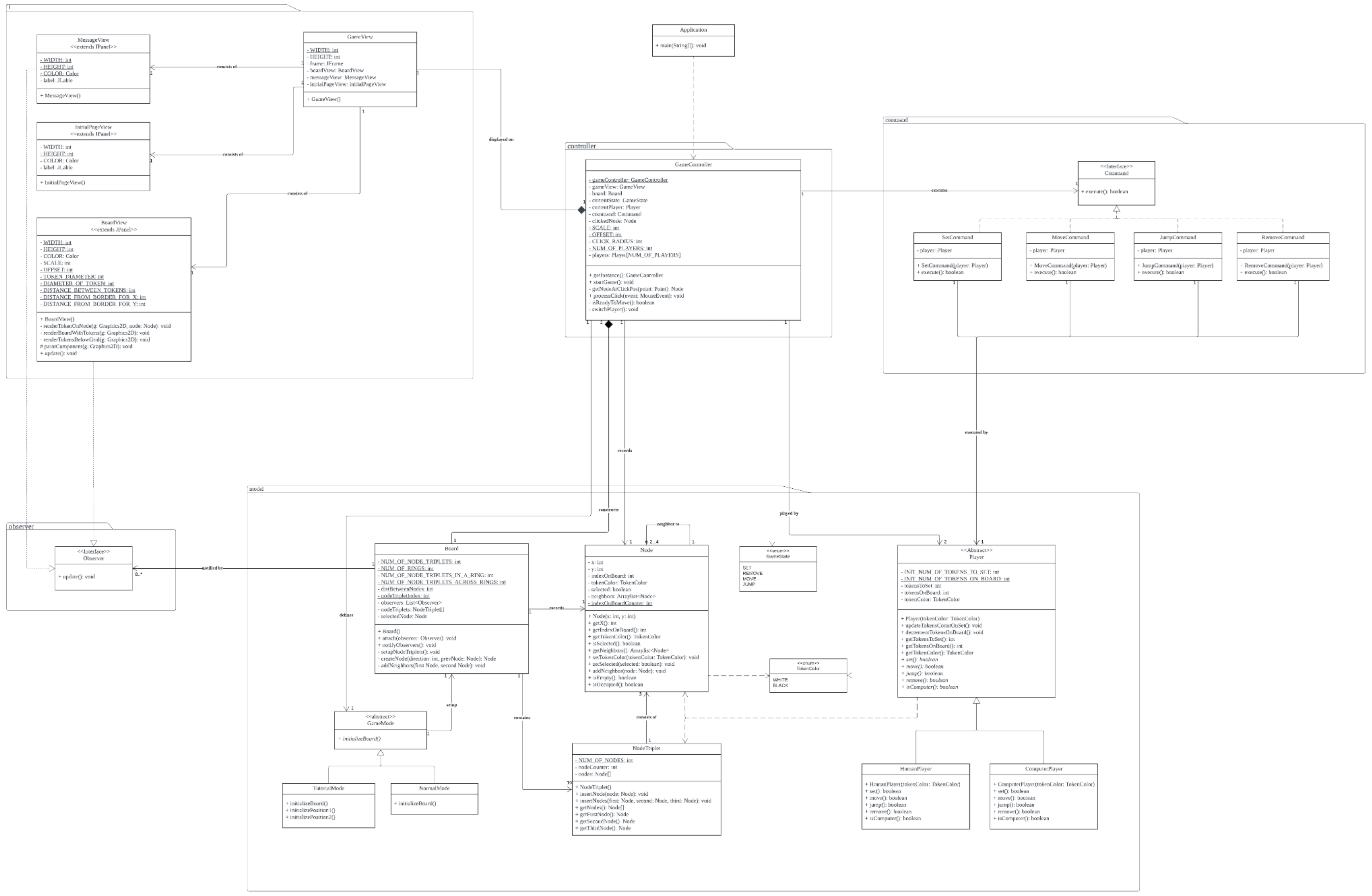
Team Details:

Huang GuoYueYang 32022891 ghua0010@student.monash.edu
Kuah Jia Chen 32286988 jkua0008@student.monash.edu
Tee Shun Yao 32193130 stee0005@student.monash.edu
Ong Di Sheng 31109667 dong0009@student.monash.edu

Table of Contents

Class Diagram

Lucidchart link



Design Rationales

1) Design Patterns Implemented and Alternatives that were Discarded

In this section, we will discuss why we have applied a particular design pattern for your software architecture and why we ruled out two other feasible alternatives.

Architecture Pattern

1. MVC

The main structure of the game uses model-view-controller architecture ***MVC***. View refers to the part which handles the interaction between the user and the interface. Classes like GameView, BoardView, MessageView and InitialPageView define how the interface looks like. A good view is needed to display and showcase a good application as it is what the users are playing on. It defines what would be responded to by the users in the interface. The model defines the game logic. It has core components like a board, node, player, tokenColor, gameState and so on. It is how the game works. This part is crucial as the game logic and rules are written as code that needs to be completely correct. Any inconsistency can cause a bug and severely affect the balance of the game. By separating the internal logic and external interface, we can manipulate the game easier. To ensure correctness, we have to make sure our model is 100% correct. To ensure user experience, we can focus on the view to improve the display of the game. The controller part controls the interaction between the model, view and user. It handles the request of the user to read or change the data of the system. In our case, it will process mouse clicks, obtain the node clicked, switch player turns, check the game state and so on. It will call the method of the model to run the game and inform the view on what to display. By having MVC, we manage to separate the responsibility and make the code easier to be maintained.

Design Pattern

1. Observer

As the game requires the UI to be rerendered when the user interacts or clicks on any empty position of the board, we initially thought of a way to use ad hoc methods to update the UI by periodically checking for changes or manually triggering an update from the controller. However, this approach can be inefficient and may lead to a delay in rendering updates on the UI. Thus, we choose to implement the observer pattern as it allows the UI component to register itself as an observer to the model in which the UI(BoardView) will be notified automatically whenever there is a change to the board pieces(set token) in real time. At the same time, the observer pattern also enables us to adhere to the ***Open-Closed Principle*** where we can introduce new observer classes in the future without having to change the code of the game model.

2. Singleton

The singleton design pattern restricts a class's capability to be instantiated into a single object that may be accessed from anywhere via a public method. To keep track of the model and display operations in our game, we developed a GameController class. This class manages user clicks and the description of the game's current state. We chose to adopt the singleton design since we didn't think the game needed several GameController objects. This makes sure that just one instance of the class is made and grants instant access to the entire world.

3. Command

The command pattern is a behavioral design pattern that transforms a query into an independent object that encapsulates all the necessary information. According to the command pattern, GameController objects shouldn't carry out operations like moving or setting on their own. Instead, all the implementation information should be extracted into a separate Command interface class with a single function to execute the command within each concrete class that implements the command. This allows the GameController object to easily initiate the command and let it handle the specifics. This approach reduces coupling between the GameController and the command logic layers, which adheres to the ***Open-Closed Principle***. This design pattern also enables the easy assembly of simple commands into more complex ones. The command design pattern thereby offers a practical middle layer that encourages code reusability and maintainability, enabling the introduction of new commands without causing old code to malfunction.

Considered Alternatives that Were Discarded

1. Strategy

We originally intended to develop an interface called Strategy with two classes—HumanStrategy and ComputerStrategy—to introduce the Strategy design pattern. We chose this strategy because the algorithms used to calculate moves for human and computer players differ. The Open-closed approach was followed in order to keep the implementation details of the algorithm distinct from the code that calls it and to make it possible to introduce new strategies without changing existing code. But after more deliberation, we decided that this step wasn't essential. We came to the conclusion that the HumanPlayer and ComputerPlayer classes themselves could include these logics, negating the necessity for the design pattern in our design.

2. State

A behavioral design pattern called the State enables an item to change its behavior when its internal state does. After giving it thoughtful consideration and investigation, we have decided to apply the Command design pattern instead of the State design pattern. We found that different kinds of operations, like moving or placing a token on the board, required different sorts of parameterization and encapsulation in our code. The need to separate these operations from the objects that perform them led to the creation of the Command pattern. In contrast, the State pattern functions better when an object's behavior changes depending on its internal state, such as when a video game character has different states depending on its present condition, such as walking, jumping, or attacking. Although both patterns can be used to depict behaviors, we discovered that the Command pattern was more suitable for our specific needs.

2) Design Rationale for Classes

In this section, we will discuss our decisions around inheritance, why we decide to use (or not use) a particular class, and why our decision is justified in our design. Additionally, we will explain how we arrived at the two sets of cardinalities, such as why we chose 0..1 instead of 1..2. These cardinalities are particularly relevant in the GameView and Board sections.

1. BoardView

The BoardView class is a critical component of the application's front-end, responsible for rendering the game board using Java Swing. To achieve this, it extends the JPanel class to set the size and color of the board. Additionally, it implements the Observer interface, allowing all UI components to register as observers of the model. This enables the BoardView to receive automatic notifications whenever there is a change to the tokens, ensuring that the board is always up-to-date in real time. By leveraging these powerful mechanisms, the BoardView class provides a seamless and dynamic user experience for players. The BoardView class is only responsible for showing the user the game board in accordance with the ***Single Responsibility Principle***. Additionally, since each method in this class performs either a command or a query action, but not both, we obey the ***Command-Query Separation Principle***. Each constant used in this class is kept as a private constant class attribute to increase readability and adhere to the "***avoiding excessive use of literals***" principle.

2. MessageView

The MessageView class is a crucial component responsible for the front-end display of information using Java Swing. As an extension of the JPanel class, it provides the ability to set the size and color of the message panel. All essential messages, such as information regarding the current game, are conveniently displayed on this panel to notify the user in real time. The MessageView class was designed in accordance with the ***Single Responsibility Principle*** to display information relevant to the current state of the game. We have adhered to the principle of "***avoiding excessive use of literals***" by storing the constants used in this class as private constant class attributes, which improves the readability of our code.

3. GameView

The GameView class is responsible for creating the user interface using Java Swing. It combines the BoardView and MessageView classes to provide a complete interface for users to play the game. This class has a ***cardinality of 1:1*** to both BoardView and MessageView, indicating that it contains one instance of each. The GameController instance is displayed within the GameView with ***cardinality 1:1*** as one GameController displays one GameView, while the BoardView class displays the game board and is dynamically updated whenever the position of tokens is changed. The MessageView class displays a message panel below the game board to provide important information to the user. According to the ***Single Responsibility Principle***, this class's sole responsibility is setting up the user's game display. Furthermore, none of the methods perform both command and query operations in accordance with the ***Command-Query Separation Principle***. Additionally, each constant in the GameView class is stored in a private constant class attribute, based on the principle of "***avoiding excessive use of literals***" to improve code readability.

4. Observer

The observer class in Java defines an interface that serves as a blueprint for other classes to implement a set of required methods. One such method is the 'update()' method, which is invoked to notify the BoardView class of any changes in the user interface elements. Upon receiving the notification, the BoardView class can re-render the game board with the updated UI elements. This ensures that the changes made in the UI elements are reflected correctly on the game board. We can see that every method in this interface is either a command or a query, following the ***Command-Query Separation Principle***. Additionally, adhering to the ***Interface Segregation Principle***, we only implement the necessary methods.

5. Board

The Board class is an integral part of the game system as it creates and assigns indexes to all the nodes. Whenever there is a change in the UI elements, such as tokens, the BoardView is notified to update and render the game board in a visually appealing way. On the other hand, the GameController class constructs the game board and enables the user to play the game seamlessly. It is noteworthy that the GameController has a ***cardinality of 1:1***, indicating that a single GameController constructs one Board, thereby ensuring that the game operates as expected. To ensure adherence to the ***Single Responsibility Principle***, this class is solely responsible for managing the game logic related to the game board. Furthermore, none of the methods do both command and query operations in order to adhere to the ***Command-Query Separation Principle***. Furthermore, we adhere to the rule of "***avoiding excessive use of***

literals" by keeping each constant as a private constant class attribute in the Board class, which makes our code easier to read.

6. Application

Application class acts as the main class which runs the program. It will get an instance from the gameController and start the game. This complies with the ***Single Responsibility Principle*****,** where the class only has only one simple functionality.

7. Player

The player is defined as an abstract class to provide abstraction. The implementation abides by the ***Open-Closed Principle***. The player class is opened for extension and closed for modification. The spectrum of what a player can do are fixed, this includes set, move, jump and remove. Any class that inherits this class cannot add other capabilities other than the fundamental capabilities of the player. However, they can extend and override abstract methods to define how different types of players will perform each action. The attributes and methods that apply to all types of players are defined with concrete implementation, this applies the ***Don't Repeat Yourself Principle.*** This also adheres to the principle of ***classes should be responsible for their properties.*** In the class, it records the number of tokens that need to be placed on the board, which is different for each player in a different situation. This requires a counter for each player, thus it is put in the player class. Each player will be assigned a color, white or black. Thus ***Avoiding the excessive use of literal*** is applied too. During the initialization, the number of tokens to be placed and the number of tokens on board are 0 and 9 respectively. Instead of just using the value in the constructor, we declare two static final constant variable INIT_NUM_OF_TOKENS_TO_SET and INIT_NUM_OF_TOKENS_ON_BOARD to store the initial value since it is consistent throughout the class and game.

8. HumanPlayer

HumanPlayer will inherit the Player class. It will override all the relevant methods, the implementation details are defined here where the user input, clickedNode will be used to navigate the moves. The class has a dependency with node and nodeTriplet classes. Node triplet is referenced to loop through all the nodes on the board. Functions in the node class are called to put or remove a token from the node.

9. Node

Node class refers to the position on the board. It is also used to record the position of a token. If there is a token on the node, the tokenColor attribute of the node will be set to either white or black, indicating the presence of the white or black node. If the attribute is null, there is no Node. It has an association with itself, with a ***cardinality of 1 :2…4***. It is an ArrayList recording all the neighbouring nodes. A static counter is used during the initialization of the board and node, where each node will be assigned a unique id. Since gameController and Board classes record the node selected by the system or player, they both have a node attribute, with a ***cardinality of 1:1***.

10. NodeTriplet

NodeTriplet is a class that is defined to have 3 nodes that appear in the same row in a set. It will be used to check the mill in the implementation of the next phase. Thus it has a ***cardinality of 1:3*** with the Node class. The class upholds the ***Single Responsibility Principle*** as it serves as an object to store a set of three nodes. Other than the necessary basic functions, there are no other responsibilities for the class. Since a board has 16 sets of node triplets, the board has an array storing the node triplets, with a ***cardinality of 1:16***.

11. TokenColor

TokenColor is an enumeration of the color of the token. There are 2 colors here, white and black. The class follows ***to avoid the excessive use of literal***. Having this class makes it consistent across all classes on how we define the color of tokens or nodes.

12. GameState

The GameState class is an enumeration that stores constants used to indicate the current game state during each iteration. The GameController relies on these constants to determine the current state of the game and execute an appropriate action. Using an enumeration class with constants helps us ***avoid the excessive use of literals***, and we don't have to create any local attributes within the GameController class to indicate the current game state. This implementation provides flexibility for future development; we can easily add more game states by adding new constants to the GameState class. This approach adheres to the ***Single Responsibility Principle***, as the enumeration class is only responsible for storing constants used during the game implementation.

13. GameController

The Board and GameView of the game, as well as the processing of user interactions, are all managed by the GameController class. As a result, it creates a ***composition*** relationship between the Board and View and the GameController class by holding instances of both. The GameController class generates the game and executes the appropriate actions in response to user clicks. The ***cardinalities*** between Board and View with GameController has been discussed in the corresponding section. Additionally, it stores references to Player instances to check the current game status, including which player's turn it is and who can perform the move command after placing nine tokens on the board. There is an ***association*** relationship between the GameController and Player classes as a result, with a ***cardinality of 1:2.***

The GameController also keeps a Command instance that serves as a representation of the game's current state and aids in carrying out the appropriate logic. The GameController keeps track of the node that is now being clicked in order to determine which node is being clicked and execute the proper game logic, creating an ***association*** between them. The class makes use of the ***Singleton*** design pattern to guarantee that there is only one GameController instance throughout the whole game. This approach ensures consistency and prevents the GameController from being duplicated unintentionally. In addition, we have stored each of the constants in the GameController class as private constant class attributes. This approach aligns with the principle of "***avoiding excessive use of literals***", which enhances the readability of our code.

14. Command

The Command class is an interface that enables the use of the ***Command*** design pattern in our system. As discussed in the previous section, this pattern brings various benefits to our design. Since we may need to introduce additional types of commands in the future, we have implemented this interface to allow for extension. This approach aligns with the ***Open-closed Principle***, which emphasizes the importance of designing systems that are open for extension but closed for modification. Additionally, we can observe that all the methods in this interface are either a command or a query, but not both, which complies with the ***Command-Query Separation Principle***. Furthermore, by implementing only the relevant methods, we are following the ***Interface Segregation Principle***.

15. MoveCommand

The MoveCommand class implements the ***Command*** interface and its main responsibility is to execute the move token logic provided by the Player class. This is achieved by storing the corresponding ***Player*** instance and calling the relevant function when the MoveCommand is executed, which creates an ***association*** between them. Adhering to the ***Single Responsibility Principle***, this class solely focuses on the task of moving a token on the board from one position to an adjacent one. Additionally, we maintain the ***Command-Query Separation Principle*** as none of the methods perform both command and query operations.

16. SetCommand

The SetCommand class implements the ***Command*** interface and its main responsibility is to execute the set token logic provided by the Player class. This is achieved by storing the corresponding ***Player*** instance and calling the relevant function when the SetCommand is executed, which creates an ***association*** between them. Adhering to the ***Single Responsibility Principle***, this class solely focuses on the task of placing a token onto a particular position on the board. Additionally, we maintain the ***Command-Query Separation Principle*** as none of the methods perform both command and query operations.

16. JumpCommand

The JumpCommand class implements the ***Command*** interface and its main responsibility is to execute the jump token logic provided by the Player class. This is achieved by storing the corresponding ***Player*** instance and calling the relevant function when the JumpCommand is executed, which creates an ***association*** between them. Adhering to the ***Single Responsibility Principle***, this class solely focuses on the task of allowing a token on the board to jump from one position to another one. Additionally, we maintain the ***Command-Query Separation Principle*** as none of the methods perform both command and query operations.

17. RemoveCommand

The RemoveCommand class implements the ***Command*** interface and its main responsibility is to execute the remove token logic provided by the Player class. This is achieved by storing the corresponding ***Player*** instance and calling the relevant function when the RemoveCommand is executed, which creates an ***association*** between them. Adhering to the ***Single Responsibility Principle***, this class solely focuses on the task of removing a token from a particular position on the board. Additionally, we maintain the ***Command-Query Separation Principle*** as none of the methods perform both command and query operations.

3) Design Decisions: Key Classes and Relationships

In this section, we will discuss two key classes that we have debated as a team, e.g. why was it decided to be made into a class, and why was it not appropriate to be a method. Additionally, we will explain two key relationships in your class diagram, e.g. why is something an aggregation not a composition?

Key Classes Explanation

1. One key class is the node class. Instead of having token class which refers to the actual token played by the player, we define the presence of token using node with an attribute of tokenColor. If the tokenColor attribute is white or black, it shows that there is a white or black token above it. The setting and checking the number of nodes are checked using counter. The ideas emerge from the reasoning that token has no responsibility. It just needs to be recorded, and there is no capability that a token possess. If there is extra responsibility in the coming sprints, we can add the token class. For this phase, we decide to keep it simple and not overextend the design as we know that it is a trade-off.

2. Another key class is Board class. In the MVC model, we will delegate the tasks of handling game logic in the model part, more specifically, the Board Class. Initially, we are thinking about the possibility of leaving the logic to gameController or creating a new class called gameEngine. The game logic includes checking a mill, a winning condition, the condition of jumping and so on. In the end, we came up with using the Singleton design pattern on the board class. It is a class which knows all the positions of the tokens, thus it is the most suitable class to handle the task. GameController acts as the controller and works between model and view. Therefore, it is not suitable for handling the game logic.

Key Relationships Explanation

1. The relationship between the GameController class and GameView class is a ***Composition*** as the GameController class would be the owner class, and the GameView class would be the composited class. The reason why it must be ***Composition*** instead of ***Aggregation*** is that the GameController class has a strong relationship with the GameView class, as it creates an instance of the GameView class inside its constructor. The GameController class is responsible for the creation and destruction of the GameView object, and the GameView object cannot exist without the GameController object.

2. The relationship between the Board class and Observer interface is ***Association*** instead of ***Dependency*** because the constructor of the Board class stores a list of Observer objects in order to make sure the BoardView class is notified to update and render the game board in a visually appealing way in real-time.