FIT3077: Software Architecture and Design

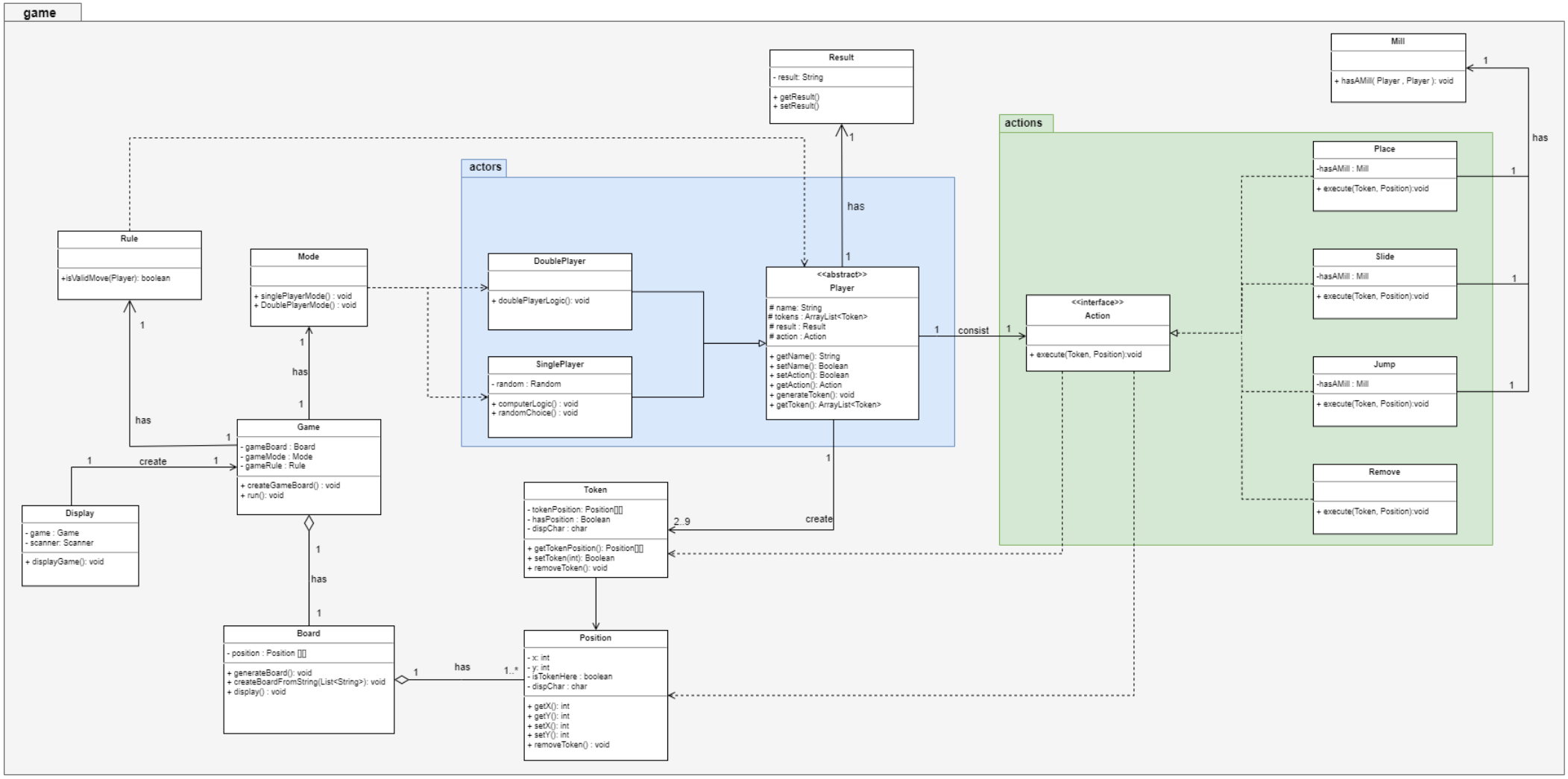
Sprint 2

MNE Tech ©

Ethel Lim
Nethara Athukorala
Mahesh Suresh

Table Of Contents

Class Diagram



Architecture and Design Rationale

Game package

All the classes are placed into a game package since they are part of the game.

Actors package

The actors package consists of the classes that are related to the user/player. The classes within the actors package include: Player abstract class, DoublePlayer and SinglePlayer classes.

Player

Player abstract class is created for setting the common attributes and methods, then defining abstract methods to allow different implementations from classes that extends to Player, such as DoublePlayer and SinglePlayer. It is created as an abstract class to avoid violating the “DRY " principle and make use of the open-closed principle, so that the abstraction benefits the code to open for extension and close for modification.

Additionally, Player has an association relationship 1 - 2..9 to Token class, as each player was given 9 tokens at the start of the game. Then the token will be gradually removed as the game progresses, until a player has only 2 tokens remaining, where the game will end.

Double Player

The DoublePlayer class extends to the Player abstract class to inherit and reuse the methods and attributes created in the Player abstract class. Implementing inheritance with the Player class ensures that no code is repeated hence the DRY principle is not violated. Then it implements the abstract methods defined in the parent class. This class is responsible for coding the logic when the user chooses to play as a double player.

SinglePlayer

The SinglePlayer class extends to the Player abstract class and follows a similar pattern as the DoublePlayer class, except that it is responsible for coding the logic for a single player.

Actions package

The actions package is concatenated with classes that are related to the action performed by the token. The following classes are present in the actions package: Action, Place, Slide, Jump, Remove classes.

Action

The Action class is the top-level class that encompasses all the possible actions that can take place in the game. It has a dependency relationship with the Token and Position classes. As the action involves the player choosing a token and moving it to a new position in which each action leads the token to a different position on the board. Since it has a dependency

relationship with the two classes, the Action class is decoupled from the specific implementation of the game logic. This decoupling makes it easier to maintain, as changes made in other classes wouldn't affect the Action class.

The Action class was made into an interface as there were a number of actions with different behaviour to be performed by the player. This allows more flexibility, as different actions can implement this class to have their own implementation way and own function.

Place

The Place class implements the Action class since it is an action that executes when the player chooses to place the token on a specific position. Hence, it can implement the abstract method 'execute' defined in the Action class. It also has a 1 to 1 association relationship with Mill class to check if a player has a mill.

Slide

Slide class is one of the actions that can be performed by the token when all tokens are placed on board, so it implements Action class. It limits the position where the token can be moved to. It has a 1 to 1 association relationship with Mill class to check if a player has a mill.

Jump

Jump class is an action to be performed by the token whenever the player is left with 3 tokens on the board. Hence, it implements the Action class to implement the abstract method defined. The Jump class represents the action of jumping a token to a valid position on the game board regardless of lines or adjacency. It also has a 1 to 1 association relationship with Mill class to constantly check if a player has a mill.

Remove

Remove class is an action that removes the token of the opponent player from the board. Whereby this action will execute whenever the player has a mill. Since it is also an action, it also implements the Action class to implement the method.

Mill

Mill class is an important class that checks if a mill is formed by a player. Player has a mill whenever they form three tokens in a row along a straight line on the game board. This class provides methods for checking if a mill has been formed. A separate class was created for the mill instead of having a method to make the code more modular and adhere to the Single Responsibility Principle, as the responsibility of checking mills is isolated to this class.

Game

The Game class is the main class of the Nine Men's Morris game and it is responsible for coordinating the different components of the game, such as the board, the rules, and the game mode. Therefore, this class has an association relationship 1 to 1 with the Mode and Rule classes and an aggregation relationship 1 to 1 with the Board class to allow it to coordinate

the different components of the game. It has an aggregation relationship with the Board class since it must instantiate a Board class to start the game, while the Board class can still exist on its own.

Display

Display class is created to the final outcome of the Nine Men's Morris game which allows the user to interact with the gameboard. This class will therefore have a 1 - 1 relationship with the Game class as the display will only need one game instance, since only one game was needed throughout the program.

Board

The Board class is created as a Singleton pattern whereby only one instance of Board class can be created. This was made to avoid multiple instances of Board class, as the game can only have one game board. Hence, the class will be responsible for initialising the board UI and displaying the current board status.

This class has an aggregation relationship 1 to 1..* to the Position class instead of composition, as a game board is made up of multiple positions while the position does not require the gameboard to exist. This applies that the Board needs Position to exist while Position can exist without having a Board, resulting in an aggregation relationship.

Position

Position class represents each location created in the game board with the two attributes x and y. These attributes are essential in order to determine the current position of the tokens on the gameboard and for validating player moves. Hence, the class consists of methods for removing and placing tokens.

Token

Token class is created for managing the tokens used in the game. It contains several methods that allow for setting the position of a token, displaying the character, removing a token from the game board, and checking if a position is currently occupied or not. This class has a dependency relationship with the Action interface and association relationship with the Player abstract class and Position class. The Token class adheres to the Dependency Inversion Principle (DIP) by depending on abstractions rather than concrete implementations. This allows for flexibility in the implementation of the game board and ensures that the token class is not tightly coupled to any specific implementation detail. The Token class also has an aggregation relationship with the Board class, which indicates that it is a part of the Board object but still can exist independently.

Rule

The Rule class is created to set up the game rules and ensure that no illegal moves are made by the players. It has methods to check if the action obeys the game rules. This class has a dependency relationship with the Player class as it needs to be able to access player

information to enforce the game rules. A separate class was created for the Rule class instead of having a method in the Game class to avoid having “god” classes.

Result

The Result class is within the game package of the class diagram for the Nine Men's Morris game. Its main responsibility is to store the result of the game for each player. It has a string attribute to store whether the player has won or lost the game, and public setter and getter methods to update and retrieve the result. The Result class has a 1 to 1 association relationship with all the player abstract classes. This implementation allows the Result class to store the result of the game for each player separately. Additionally, this class can be extended in the future to store more information about the game, such as the score of the game.

Mode

The mode class is responsible for handling the different game modes available in the Nine Men’s Morris game, such as the single player and double player modes. It has a dependency relationship with both the SinglePlayer and DoublePlayer classes and an association relationship with the Game class. Additionally, the Mode class allows the player to choose the game mode before starting the game, this ensures that the game is customizable and can be tailored to the user’s preferences. The dependency and association relationships to both SinglePlayer and DoublePlayer classes ensures that the Mode class can handle both the game modes effectively.

Architecture

All of the classes in the class diagram adheres to the SOLID principle of Single Responsibility Principle (SRP), meaning that each class has a single, clearly defined responsibility or task to perform. Additionally, it makes the code more modular and easier to understand, as each class can be developed and tested independently.

A singleton instance as part of the creational patterns was used in the Board class. This was done as the game only requires one board in order to be carried out and by using a singleton method we are able to save resources and run the game with only one instantiation. The reasoning behind not using either the behavioural patterns in this instance is that we weren’t creating algorithms but instead captured the characteristics of the board, the behavioural patterns support iterators and more so algorithm strategies that weren’t relevant in our given scenario.

Moreover, the strategy design pattern which is part of behavioural design pattern is applied to the actions performed by the tokens. As actions are similar but they execute in a different behaviour, hence an Action interface is created to be able to implement the actions in different ways.