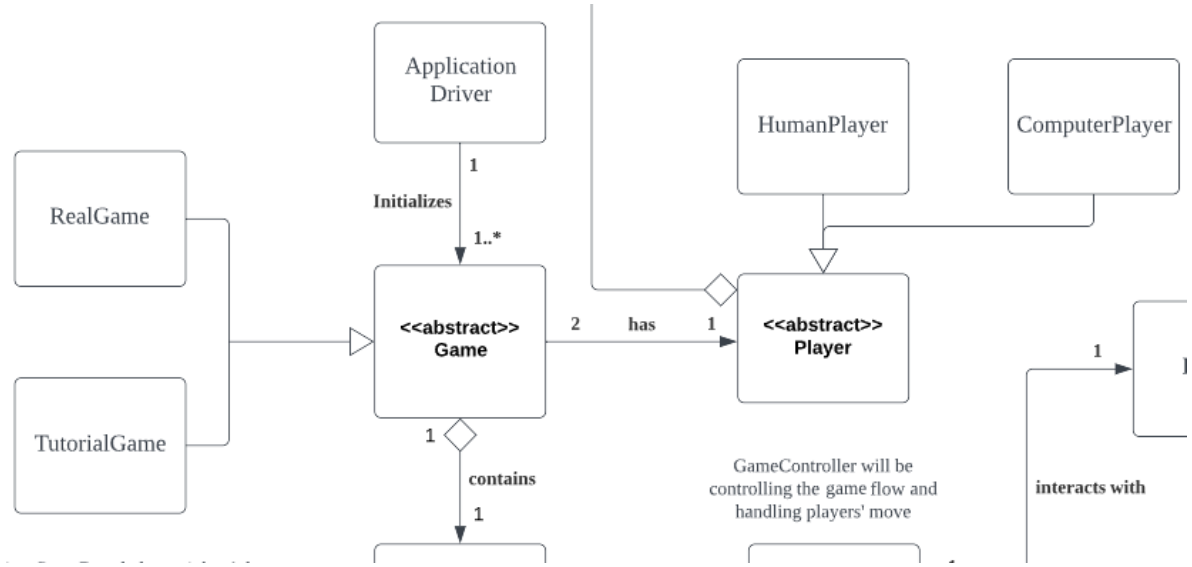
Assignment Sprint 1 - Rationale for Basic Architecture

Last Updated: 03/4/2023
Authors: Muhammad Abdullah Akif, Subhan Saadat Khan,
Ong Li Ching, How Yu Chern

Game and Players - Abstraction



The Game class represents an instance of a single 9 Man Morris game, where Application Driver initialises the entire program since the main Java method: Public Static Void Main is located there that creates the Game. The Application Driver can run multiple instances of Games (though typically it would be 1 game at a time). We have made Game Class an abstract class that is extended by 2 types of games: RealGame and TutorialGame, to support a part of Advanced Requirement A where players have the option of playing a tutorial game to learn the rules of 9 Man Morris.

The Game Class has a 2 to 1 association relationship with the abstract class Player, since a 9 Man Morris game has a fixed number of two players. The Player Class is extended by HumanPlayer and ComputerPlayer to allow Human Vs Human or Human Vs Computer games, supporting Advanced Requirement C.

Both the classes Game and Player with their child classes support Liskov Substitution Principle, as either one of the child classes can replace the parent classes. This allows the program to choose various game modes, or different player types. If a new type of gamemode or player type is introduced, it can be easily added by extending the parent and inheriting common methods and attributes whilst implementing their own functionality.

Gameboard, GameManager, Display, Positions, and Pieces.

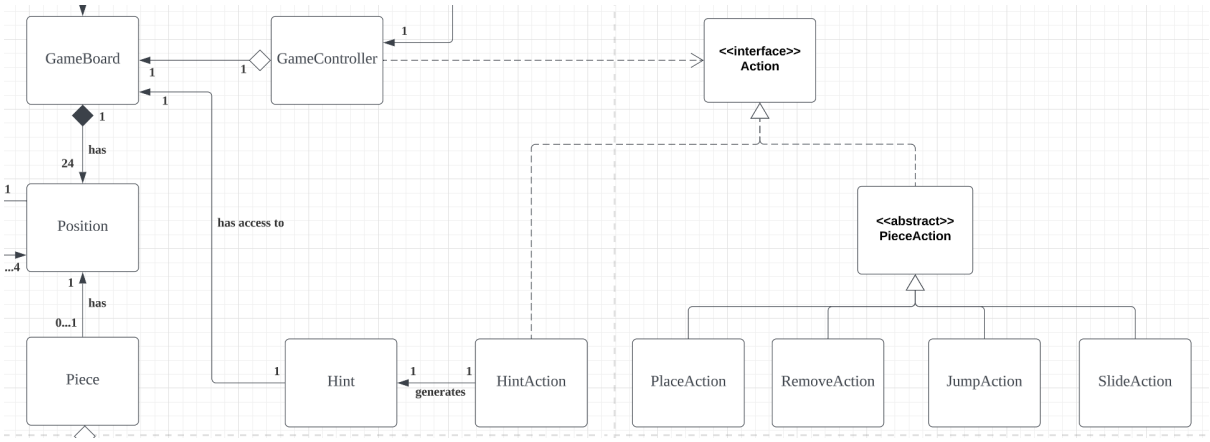
In a Game, a GameBoard Class represents a generic 9 Man Morris board, which can be modelled as a Graph with 24 interactable positions that could be empty or contain a piece. Each position will know 2 to 4 of its neighbouring positions, which is important for determining possible moves for pieces to slide to, and processing hints.

GameBoard will also have an association to a GameManager, a class responsible for mediating inputs from the screen (Display Class), and the actions each player takes, so that the GameBoard can be updated appropriately. GameManager class distributes the responsibility from GameBoard to enforce single-responsibility principle - so that GameBoard will not have too many responsibilities or functions, and would be easier to manage and extend.

Likewise, a piece would know its position when it is on the board, in order to track each piece's state when it moves on the board. If a piece has no positions, it indicates that it has not yet been placed onto the board, or has been removed. Each piece is also assigned a standard colour from an Enum: Color, which represents which player the piece belongs to.

Positions have a composition relationship with the Gameboard, as if the Gameboard is deleted, the positions lose meaning since they are relative to a vertex on the Gameboard Graph. Meanwhile, Colours has an aggregation relationship with piece as it can exist independently without piece, and can be reused as a standard for colouring other elements in the game if required.

Actions and Hint



There are 5 actions that can be taken by the players when playing the game:

1. Place a piece onto the board
2. Remove an opponent's piece when they form a mill
3. Jump a piece when a player only have 3 pieces left
4. Slide a piece along the lines on the board
5. Click on a hint option to view all the legal next moves (for Advanced Requirement A)

We have created a class for each of these actions, which are PlaceAction, RemoveAction, JumpAction, SlideAction and HintAction.

We have also created an **abstract class PieceAction** for PlaceAction, RemoveAction, JumpAction and SlideAction since they involve manipulating pieces on the board. By creating this abstract class, we can define the common functionality and methods that are shared by all the subclasses. For example, these actions need to check whether the action is valid at the current state of the game board. We can create an abstract method isValid() that checks the validity of the action at the current state of the game board. Since it is abstract, this method must be implemented by each of the concrete classes. In this way, we can avoid duplicating code across the concrete classes and ensure they all conform to a consistent interface, following the **principle of abstraction**. Moreover, in the future, we can create new concrete classes that inherit from PieceAction. The new concrete classes can implement their own logic without the need to modify PieceAction. This follows the **Open/Closed Principle**.

The Hint class represents the hint option in the game and is responsible for generating all the legal moves the player may make as their next move. **HintAction** has a **1 to 1** relationship with **Hint**. It is responsible for displaying the list of the legal moves generated by Hint to the player when they click on the hint option. This design follows the **Single Responsibility Principle**, as each class has a clear and specific responsibility where Hint generates legal moves and HintAction displays them to the player. This makes the code more modular and easier to maintain.

The **Hint** class has a **1 to 1** relationship with **GameBoard** because it needs access to GameBoard in order to generate a list of all the legal moves. This approach follows the **principle of composition** which allows modularity and code reuse. For example, GameBoard can have a method to get all the adjacent positions of a position, since GameBoard is responsible for maintaining the game state, including the positions of all the pieces on the game board. Hint can use this method to generate legal moves. Therefore, Hint does not need to replicate the functionality of GameBoard.

We have created an **interface Action** which defines methods that are specifically for actions that can be performed in the game. **HintAction** and **PieceAction implements Action** as they are actions that can be performed in the game.

We choose to implement Action as an interface because interface allows greater flexibility and extensibility as a class can implement multiple interfaces but can only extend from an abstract class.

By implementing the Action interface, GameManager can perform actions on the game board without knowing the implementation of each action. For example, the Action interface might have a method called execute() which would perform the action on the game board, and PlaceAction would provide implementation of the method. When the player places a piece, GameManager would call this method on the Action object to perform the action, instead of directly on PlaceAction. This design follows the **principle of encapsulation** where only Action is exposed to GameManager. All the implementation details of

PlaceAction are hidden from GameManager. This also allows us to maintain the code easier as change to PlaceAction will not affect GameController to perform the action.

Furthermore, the use of Action interface follows the **Interface Segregation Principle**, since it only includes methods that are related to actions. In addition, adding new actions only requires implementing the Action interface and its methods without affecting existing code. This follows the **Open/Closed Principle**.