Australian National University

COMP6710 Structured Programming Design Report

Authors:

Xu YiWen (UID: u7799354)

Guo JingTong (UID: u7897820)

Ye Yang (UID: u7890217)

Design Idea Description

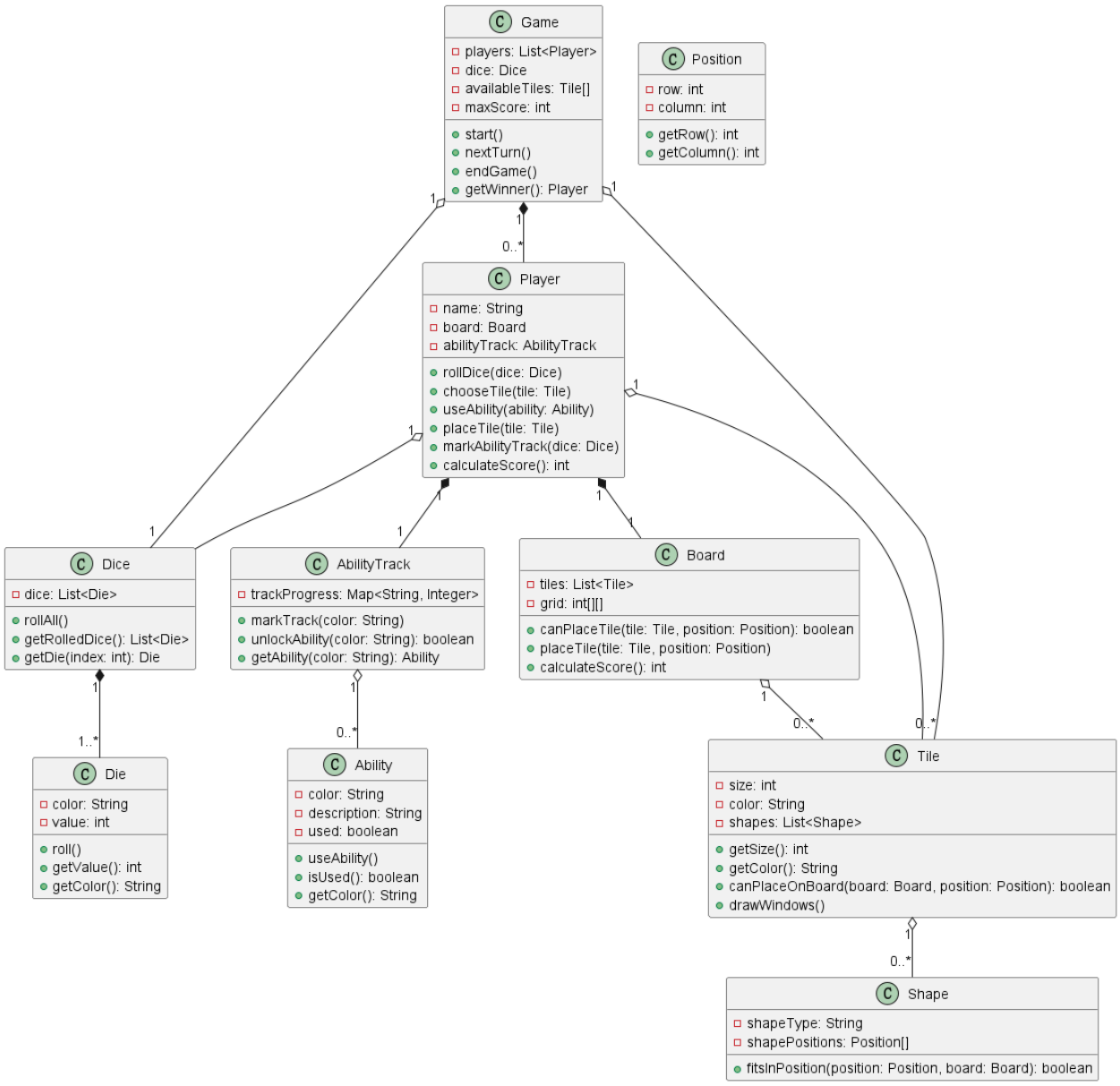

Figure 1: Class Diagram for the Game Design

1. Game Class

In our code structure, the `Game` class is the control center of the entire game. It is responsible for managing the start of the game, controlling the flow of the game (e.g., opening the next player's turn), determining the end of the game, and determining the final winner. Inside

this class, it should contain a list of all players, dice objects, and available tiles. During each turn, the `Game` class needs to instruct the current player to perform actions and coordinate interactions between players.

Role: The `Game` class controls the overall flow of the game by coordinating player turns. It manages player dice rolls, tile selection, tile placement, and ability use.

Collaboration: During each turn, the `Game` class needs to call each player's action methods, such as `rollDice()` and `placeTile()`, in turn, and update the game state based on the results of the dice rolls and the players' choices.

Reason: Centralize the core logic of the game in one class to ensure the consistency of the game flow and rules.

2. Player Class

The `Player` class represents a player in the game, and each player manages a personal board (`Board` class), an `AbilityTrack` class, and actions that are unique to the current player. Each player can roll dice, choose tiles, place tiles, and use special abilities as needed during their turn. The `Player` class ensures that each player manages their own game state independently while interacting with other players, while other players can build up their own special ability bars based on the color of their remaining tiles during their turn.

Role: The `Player` class encapsulates all of the player's behavior, including rolling dice, selecting tiles, placing tiles, using abilities, and calculating scores.

Collaboration: Works closely with `Board` and `AbilityTrack` to ensure that every action a player takes affects their game state and score.

Reason for choosing: Players are at the heart of the game, and encapsulating player behavior and state in a single class helps to clearly manage each player's actions and ensure

a fair game.

3. Board Class

The `Board` class represents each player's personal board and is used to manage the tiles (`Tile` class) placed by the player and their layout. The `Board` class is designed to ensure that each tile's placement conforms to the rules of the game (it does not overlap and must be connected to the bottom of an existing tile or building).

Role: The `Board` class manages the position and layout of the tiles placed by the player on the board and calculates the score after the tiles are placed.

Collaboration: Working with the `Tile` class, the `Board` class is responsible for checking that the placement of tiles is legal and drawing windows after placement.

Reason: According to the game rules, each player should have his own board, and the independent `Board` class makes it clearer to manage the layout of each player's board, and simplifies the operations related to tiles.

4. Dice and Die Classes

In our code structure, we need a class to manage multiple `Die` objects, so we create the `Dice` class. It not only manages multiple `Die` objects, but also provides the functionality to roll the dice and get the result. Among other things, the `Die` class represents a die with two attributes, color and point. These classes allow the game to simulate a dice roll and pass the result to the player for decision making.

Characters: The `Dice` class generates the color combinations available to the player during the turn via the `rollAll()` method. Each `Die` object represents the color and value of one die.

Collaboration: During each turn, the results of the `Dice` rolls directly influence the decisions made by the `Player` class, such as which color tiles to choose or which ability tracks to mark.

Reason for choosing: A separate `Dice` class simplifies the handling of randomness in the game and provides players with a variety of choices.

5. Tile Class

In our code design, the `Tile` class represents the tiles in the game. Each tile has its own size and color properties and may have a specific shape (`Shape` class). This class manages the properties of the tiles and ensures that they can be legally placed on the player's board.

Role: The `Tile` class encapsulates the size, color, and shape properties of tiles and provides methods to check whether tiles can be placed on the board.

Collaboration: Working with the `Board` class, the `Tile` class provides methods for placing tiles and drawing windows to ensure that tile placement conforms to the rules.

Reason for choosing: Separating the logic of tiles into a single class makes the placement and management of tiles more flexible and simplifies the handling of the board layout.

6. Position Class

Since each tile needs a specific position to be displayed on the board, we design a `Position` class to represent a specific position on the board, which consists of two attributes: row and column. With this class, we can simplify and standardize the position handling when placing tiles.

Role: The `Position` class helps to standardize the position of tiles on the board, simplifying the process of placing and checking them.

Collaboration: Used in conjunction with the `Board` and `Tile` classes to ensure the legality of tile placement.

Reason for choosing: Representing position through a separate class improves code abstraction and flexibility.

7. Shape Class

After seeing the UI interface in the readme, we realize that each tile has its own shape, such as L-shape, T-shape and so on, so we need to design a class to manage these shapes. The `Shape` class not only represents the shapes of the tiles, but also defines the relative position of the shapes on the board. It is used to check whether a certain shape can be placed in the specified position, so as to ensure the layout rules in the game.

Role: The `Shape` class manages the shapes of tiles and their positions on the board

Class relationships

Relationship between Player and Board (`"1" *-- "1"`)

In the code structure we designed, each `Player` object is associated with a `Board` object, forming a strong one-to-one relationship. The reason for this is that according to the game rules, each player should have a separate board to manage their progress and tile layout.

Player's Relationship with AbilityTrack (`"1" *-- "1"`)

When designing this set of relationships, we considered that each player should have a separate `AbilityTrack` for tracking their progress in the game and unlocking special abilities, so we associate each `Player` object with an `AbilityTrack` object to form a strong one-to-one

relationship.

Relationship between Player and Dice (`"1" o--`)

Since the rest of the players have to choose the current dice color after the current turn, there should be a weak dependency between each `Player` object and `Dice` object. This means that all players use the same set of dice in the game, but do not need separate dice instances.

Player's Relationship to Tiles (`"1" o--`)

Although players can select tiles from the pool of available tiles for placement, this does not mean that the management of the tiles depends on a separate instance of the player, so each `Player` object has a weak dependency on the `Tile` object.

Relationship between Board and Tile (`"1" o-- "0..*"`)

In the rules, players can place multiple tiles on the board and manage them appropriately according to the rules of the game, so each `Board` object can contain multiple `Tile` objects, forming a weak one-to-many dependency.

Relationship between Tile and Shape (`"1" o-- "0..*"`)

According to our research on the rules, each tile can have many different shapes, such as L-shape, T-shape, and the number of tiles that make up the shape varies, so the player can decide how the tiles are laid out on the board according to the shape. So each `Tile` object can be composed of multiple `Shape` objects, forming a weak one-to-many dependency.

Relationship between Dice and Die ("1" *-- "1..*")

`Dice` itself is used to manage `Die` objects, so each `Dice` object contains multiple `Die` objects, forming a one-to-many strong dependency. The dice group consists of multiple individual dice, each `Die` has a different color and points.

AbilityTrack's Relationship with Ability ("1" o-- "0..*")

Similarly, `AbilityTrack` itself is used to manage `Ability` objects. Players can use different abilities by accumulating `AbilityTracks` of the corresponding color, so each `AbilityTrack` object can contain multiple `Ability` objects, forming a weak one-to-many dependency.

Relationship between Game and Player ("1" *-- "0..*")

A game can have more than one player, supporting multiplayer, which means each `Game` object can contain more than one `Player` object, forming a strong one-to-many relationship.

Relationship between Game and Dice ("1" o-- "1")

In the whole game, players share a common dice, so each `Game` object is associated with a `Dice` object, forming a weak one-to-one dependency.

Relationship between Game and Tile ("1" o-- "0..*")

In the whole game, each player can choose more than one tile, and make choices according to their needs, so each `Game` object should contain more than one `Tile` object, forming a weak one-to-many dependency.