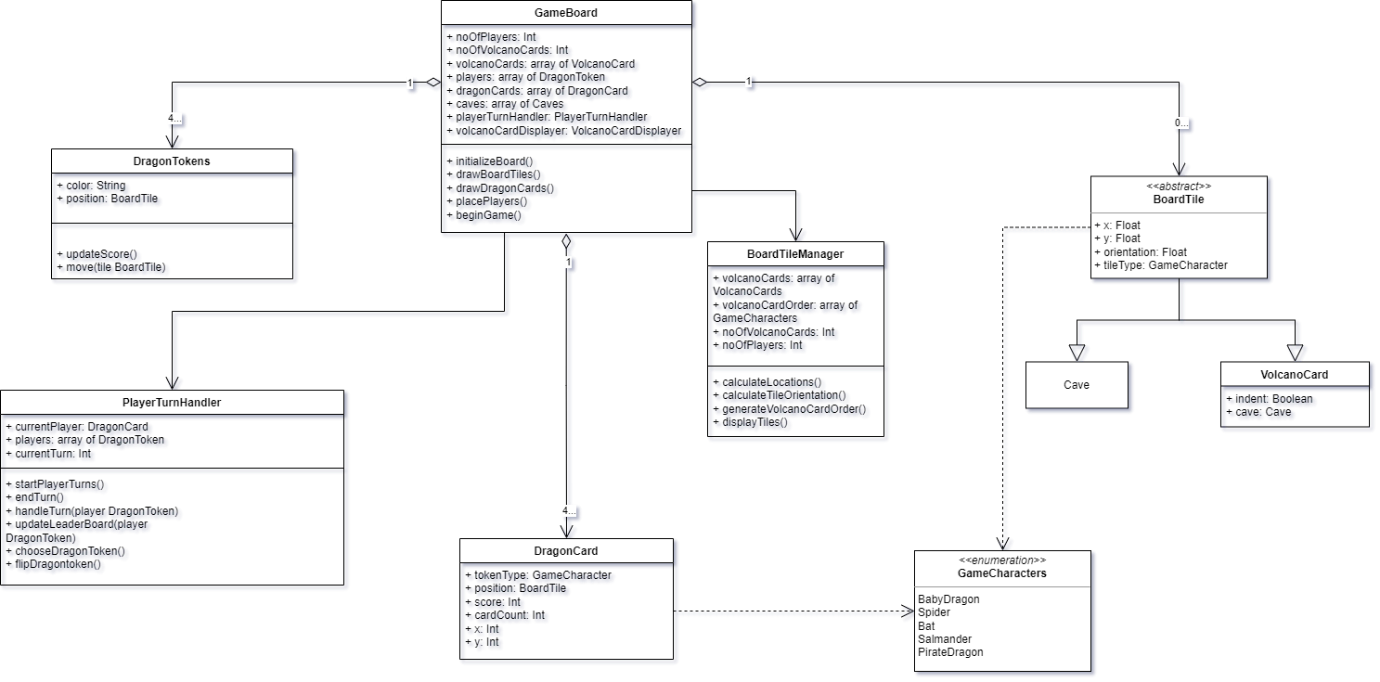
# FIT3077 Sprint 2 Object-Oriented Design and Design Rationales

## Full Class Diagram



## Key Game Functionalities

To summarize the key game functionalities:

(i) Setting up the initial game board:

* The **BoardTileManager** class is responsible for managing the game board.
* It has methods such as **calculateLocations()** and **calculateTileOrientation()** to set up the initial positions and orientations of the board tiles.
* Additionally, it has methods like **generateVolcanoCardOrder()** to randomize the positioning of dragon cards on the board.

(ii) Flipping of dragon cards:

* The **DragonCard** class represents the dragon cards (also referred to as "chit" cards).
* The **flipDragonToken()** method in the **PlayerTurnHandler** class is responsible for flipping a dragon card.
* The interaction between the **PlayerTurnHandler** and **DragonCard** classes facilitates the flipping of dragon cards during the game.

(iii) Movement of dragon tokens:

* The **DragonToken** class represents the dragon tokens on the game board.
* The **move(tile BoardTile)** method in the **DragonToken** class handles the movement of dragon tokens based on their current position and the last flipped dragon card.
* The **BoardTile** class contains information about each tile on the game board, facilitating the movement of dragon tokens between tiles.

(iv) Change of turn to the next player:

* The **PlayerTurnHandler** class manages the turns of players during the game.
* It has methods like **startPlayerTurns()** and **endTurn()** to handle the transition between player turns.
* The interaction between the **PlayerTurnHandler** class and other game components facilitates the change of turn to the next player.

(v) Winning the game:

* The conditions for winning the game are not explicitly represented in the above class diagram.
* However, winning conditions could be implemented as part of the game logic within the **PlayerTurnHandler** class or other relevant game components.
* The game could check for winning conditions based on factors such as achieving particular objectives or reaching specific game states.

Overall, the design covers key game functionalities such as setting up the initial game board, flipping dragon cards, moving dragon tokens, changing turns between players, and potentially implementing winning conditions. These functionalities are facilitated through interactions between different classes and components within the game architecture.

## Design Rationale

1. **Key Classes:**

a. **DragonToken Class:** - **Reason for Creation:** The **DragonToken** class represents the dragon tokens on the game board. It was created as a separate class because dragon tokens have unique properties and behaviors, such as movement across the board and interaction with other game components. The design promotes modularity and separation of concerns by encapsulating this functionality within the DragonToken class. - **Why Not Appropriate as a Method:** Representing dragon tokens as methods within another class (e.g., **BoardTile** or **PlayerTurnHandler**) would not be appropriate because it would violate the principle of single responsibility. Dragon tokens have their own state (position, behavior) and functionality (movement, interaction), which is best managed within a dedicated class.

b. **PlayerTurnHandler Class:** - **Reason for Creation:** The **PlayerTurnHandler** class manages the turns of players during the game. It was created to handle the logic related to player turns, including starting and ending turns, managing the sequence of players, and enforcing game rules related to turns. - **Why Not Appropriate as a Method:** Managing player turns involves coordinating multiple actions and states, such as tracking the current player, handling turns transitions, and enforcing game rules. Representing all of this functionality as methods within another class would lead to bloated and less maintainable code. By encapsulating turn handling logic within its own class, the design promotes encapsulation, flexibility, and ease of maintenance.

1. **Key Relationships:**

a. **Aggregation vs. Composition:** - **Explanation:** In the class diagram, certain relationships are represented as aggregations rather than compositions. For example, the relationship between **BoardTile** and **DragonToken** is depicted as an aggregation. - **Why Aggregation and Not Composition:** Aggregation implies a "has-a" relationship, where **BoardTile** has one or more **DragonToken**s associated with it. This relationship allows for flexibility, as dragon tokens can exist independently of specific tiles (e.g., if a token is removed from the board). Composition, on the other hand, implies a stronger ownership relationship, where the lifetime of the **DragonToken** instances is tied directly to the **BoardTile** instances. Since dragon tokens can be moved between tiles and potentially removed from the board, aggregation is more appropriate than composition.

1. **Inheritance:**
   * **Explanation:** In the provided class diagram, inheritance is not explicitly represented.
   * **Why No Inheritance:** The decision to avoid inheritance may be justified by the design's focus on composition over inheritance. Instead of using inheritance to represent relationships between classes, the design prioritizes composition and aggregation to model more flexible and modular relationships between components. By favoring composition, the design promotes code reuse, maintainability, and flexibility, as it allows for more granular control over class behaviors and relationships.
2. **Cardinalities:**

a. **0..1 Cardinality:** - **Explanation:** In the class diagram, certain associations have a cardinality of 0..1, indicating an optional relationship. - **Why 0..1:** For example, the relationship between **Player** and **DragonToken** is depicted with a 0..1 cardinality, indicating that a player may or may not have a dragon token associated with them. This cardinality is appropriate because not all players may possess a dragon token at any given time, especially if tokens are lost or not yet acquired during gameplay.

b. *1.. Cardinality:*\* - **Explanation:** Other associations in the diagram have a cardinality of 1..*, indicating a mandatory relationship where one class is associated with one or more instances of another class. - \*\*Why 1..*:\*\* For example, the relationship between **BoardTile** and **DragonToken** has a 1..\* cardinality, indicating that a board tile can have one or more dragon tokens associated with it. This cardinality reflects the game mechanics where multiple dragon tokens may occupy the same tile on the game board simultaneously.

Top of Form

## Design Patterns

**Strategy Design Pattern:**

**Why Applied:** The Strategy pattern could be applied in the design to encapsulate different strategies for setting up the initial game board. For example, the **BoardTileManager** class could have multiple strategy objects representing different algorithms for calculating tile positions and orientations. This would allow the game to dynamically switch between different runtime strategies, depending on game mode or player preferences.

**Command Design Pattern:**

**Why Applied:** The Command pattern could be applied to encapsulate actions such as flipping dragon cards and moving dragon tokens. For example, commands could be represented as objects (e.g., **FlipDragonCardCommand** and **MoveDragonTokenCommand**) that encapsulate the necessary information and logic to execute these actions. This would enable the game to support undo functionality, logging, or queuing of commands for replayability.

**Observer Design Pattern:**

**Why Applied:** The Observer pattern could be applied to notify game components about changes in the game state, such as the flipping of dragon cards or the movement of dragon tokens. For example, the **PlayerTurnHandler** class could act as the subject, notifying observers (e.g., UI components or scoring systems) about changes in player turns or game events. This would enable loose coupling between components and facilitate extensibility.

Top of Form