Design Rationale:

○ Explain two key classes (not interfaces) that you have included in your design and provide your reasons why you decided to create these classes. Why was it not appropriate for them to be methods?

GameManager class:
I have included this class as a tool to coordinate the game. Its responsibility is to start the game, manage player turns, end the game, add players to the game and create the display window. I have created a separate class to implement these functionalities. This has allowed me to enhance code organization, modularity since the GameManager encapsulates the game management logic, making it easier to maintain and extend without impacting other code sections.

PlayerMovement class:
The player movement class was created to be accountable for the managing of movement of players within the game. It updates the player's location and also checks if the player is able to move. I have encapsulated all the logic related to player movement and added it to this class, allowing for better code organization and maintainability and modularity since PlayerMovement encapsulates the game management logic, making it easier to maintain and extend without impacting other code sections.

The reasons why these two classes were not methods are the following:
The only advantages having them as a method is that we would have faster performance due to As method calls within the same class incur lower overhead and perhaps reduce code complexity due to less classes. However, the disadvantages of having them as methods outweigh the benefits.. The disadvantages include violation of the single responsibility principle which as a result would lead to a monolithic and tightly coupled design, making the code harder to maintain and extend. The advantages of having them as classes also outweigh the advantages of having them as methods because by separating concerns into different classes, the code becomes more modular, flexible, and easier to reason about. Following the single responsibility principle increases code readability, maintainability and makes the code easier to modify. Encapsulating related functionality within classes promotes code reusability, potentially facilitating usage in other parts of the application or different projects.

○ Explain two key relationships in your class diagram, for example, why is something an aggregation not a composition?

The relationship between DragonCard and Character class is an aggregation relationship. This is because dragon cards have instances of Characters, however the characters can exist without the DragonCard as we also have characters on the square of the game board. The reason why it is not a composition relationship is because if the dragon cards are destroyed the characters would still exist in the game on instance on the squares.

The relationship between the Volcano and GameManager class is an association relationship. This is because Gamemanager class has an attribute of the Volcano class signifying a structural relationship. The reason why the relationship is not a dependency is because Volcano is a part of the Gamemanger class's definition or structure, dependencies are typically temporary.

○ Explain your decisions around inheritance, why did you decide to use (or not use) it? Why is your decision justified in your design?

I did not use inheritance because each class seemed to represent a distinct entity with its own unique attributes and behaviors that were not shared with other types of objects in the game. Although the Square class and DragonCard class shared some similar characteristics such as sharing the character attribute, they had other completely different attributes and methods and functionalities, for instance the square also has an attribute to check if the player is on the square, the dragon card has nothing to do with the player as the player only moves on the square. Also there is no point in them sharing the attribute of character as the character is generated randomly and therefore the characters should be different for both the classes. There was one method in common which was to generate a random square and generate random dragon cards, however using inheritance for a method that is somewhat similar is not worth it because it will only increase the code complexity and the methods are not even the same as square generates only one random square whereas DragonCard generates 16 dragon cards.

○ Explain how you arrived at two sets of cardinalities, for example, why 0..1 and why not 1…2?

Unlike the original domain model in sprint 1, the cardinalities between the Volcano class and Dragon card is now 1 Volcano card having 1 Dragon Card, This originally used to be 1 Volcano card having 16 dragon cards, the reason this has changed is because I have given the dragon card the responsibility to generated 16 dragon cards hence adhering to the single responsibility principle and also then the volcano card would only need 1 attribute of dragon card has it can generate a list of 16 cards at random.

Moreover the relationship between square and player is that each square has 0..1 player. This is because not every square will have a player on it; some squares will have zero players. Moreover, according to the game rule each square can only be occupied by 1 player at a time, therefore the maximum cardinality is 1.

If you have used any Design Patterns in your design, explain why and where you have applied them. If you have not used any Design Patterns, list at least 3 known Design Patterns and justify why they were not appropriate to be used in your design.

I have not used any patterns in my implementation.

Singleton Pattern - creational design pattern that ensures a class has only one instance, while providing a global access point to this instance. Although we only have a single instance of the GameManager class, there is no point in providing a global access to it as it will violate the single responsibility principle and creates tight coupling between those classes and the GameManager. This violates the principle of loose coupling, which is one of the fundamental principles of good object-oriented design. Tightly coupled code is harder to modify, extend, and maintain because changes in one part of the system can potentially affect other parts.

Factory Method - a factory method is an interface that is used to create objects in a super class but allows subclasses to alter their type. The factory method could have been used to encapsulate object creation logic of different types of cards (VolcanoCard, DragonCard) making it easier to introduce new types of cards or characters in the future without modifying the existing code. However, there is no such requirement to add more different types of cards even in the extension making this redundant.

The Strategy Pattern proves beneficial when dealing with varied algorithms or behaviors required for player movement, character actions, or card effects. By defining distinct strategy classes, the game objects gain the flexibility to dynamically transition between different strategies, adapting to changes in the game state or user input. This pattern can be used for future extension but is not necessary for the sprint.