**Design Rationales**

○ 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?

GameEngine:

The Game Engine is used to process multiple different instances since it controls the other classes and produce the correct input based on the situation of the game.

Also this requires a minimum of 3 methods for these different scenarios which are not possible with one method.

VolcanoCard:

This class using the function from the square class to further create random squares, in groups of 3. This uses a init where a random square is created and then in the end it combines these 3 things into one object. The reason that this is a class and not a method is because we want the created volcano card objects to stay constant as the game runs, with a method this would not be possible since the method would be run over and over and the output would be a differing pair of square each time.

○ Explain two key relationships in your class diagram, for example, why is something an

aggregation not a composition?

VolcanoCard & Volcano is a composition relationship since the VolcanoCard’s render function is used within the Volcano classes render function this means that if the VolcanoCard object were not to exit then the Volcano also could not exit since on its basis the Volcano is made

DragonCard similarly is a composition relationship with the character class since the character is the basis of the DragonCard class. Since character is used to indicate which dragon card it is and the value is only relevant if the corresponding character is present therefore the DragonCard Class would not exist without the Character class.

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

is your decision justified in your design?

Inheritance was not required in this task since the objects created did not have any generalisation from the domain model from the previous assessment. The generalisation of character was not required since a simple basis on how the character is defined aka its init handle which image should be assigned to that character object using a match case. Other wise no classes that I used required or stemmed from any classes directly though the square was used in the Volcano card, the Volcano card and Square have completely different functionality.

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

1…2?

As mentioned above the Volcano and VolcanoCard are composition relationships. The volcano has 1 to 8 Volcano Card cardinality. This is because for every 1 Volcano there is only exactly 8 Volcano cards that are constructed at the start of the volcano initialization.

Similarly with GameEngine 1 to Volcano 1 since there can only be one volcano per game, and only one Game Engine instance is created to run the game therefore, 1 Volcano to 1 GameEngine.

Patterns:

Each class has unique properties and functions. The interactions in the game are straightforward, which explains why there are no obvious design patterns. Design patterns are frequently used to overcome common software design difficulties. Since, the game's design is straightforward enough, these patterns are not required.

For example, the Singleton pattern, which prevents a class from instantiating multiple objects, does not appear appropriate here. Since the game requires several instances of classes such as Character and VolcanoCard. The Strategy pattern, which is used when interchangeable algorithms are required within classes, is also not applicable. Because there are no interchangeable algorithms necessary for the implementation. Finally, the Observer pattern, which is beneficial for subscription methods that alert numerous objects about changes in one object, is unneeded in this architecture. Since the GameManager handles all the alerts and changes

The game emphasises direct interactions between its aspects. This is demonstrated by methods like check\_win() in the Dragon class and run() in the GameEngine class. As a result, the game's interactions may lack design patterns due to their simplicity. In conclusion, while design patterns can be useful in a variety of contexts, they are not always required or acceptable, as appears to be the case here.