FIT3077 SOFTWARE ENGINEERING: ARCHITECTURE AND DESIGN
SPRINT TWO: DESIGN RATIONALE

By: Zhan Hung Fu

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?"

TILE:

In my design, one key class is the 'Tile' class, which is an abstract class with child classes 'CaveTile' and 'NormalTile'. I decided to create these classes to represent different types of tiles that can exist on the board.

The decision to use classes for 'CaveTile' and 'NormalTile' instead of methods was based on the need to represent tiles as distinct entities with their unique properties and behaviours. For example, 'CaveTile' has additional logic for triggering the win condition when a dragon enters it, while 'NormalTile' may have different logic for handling the dragon's movement.

This design adheres to the Open/Closed Principle as I can add new types of tiles by creating new subclasses of 'Tile' without modifying the existing 'Tile' class. This promotes code reusability and allows for easy extension if additional features related to tiles need to be implemented in the game.

CHIT CARD:

Another key class is the 'ChitCard' class which abstracts the common attributes and behaviour of all chit cards, such as flipping the card and performing action. By creating this abstract class, it's easier to add new types of chit cards in the future.

'AnimalChitCard' and 'DragonPirateChitCard' classes are subclasses of the 'ChitCard' abstract class. The 'AnimalChitCard' class has an additional attribute called 'Animal', representing the animal depicted on the chit card. This attribute is used to compare with the animal on the tile where a dragon is currently located. If the animals match, the dragon will move based on the 'chit_quantity' attribute of the card.

On the other hand, the 'DragonPirateChitCard' class does not have an 'Animal' attribute because no additional checks are needed. Instead of moving the dragon forward, this type of card forces the dragon to move backwards immediately if picked. The design choice to include the 'Animal' attribute in 'AnimalChitCard' and not in 'DragonPirateChitCard' reflects the specific behaviour and requirements of each type of chit card, ensuring that each class has a clear and distinct purpose.

This design again adheres to the Open/Closed Principle as it allows for easy extension of new chit card types. Additionally, it adheres to the Dependency Inversion Principle as it allows for loose coupling between classes. The 'ChitCard' class defines abstract methods that its subclasses must implement, allowing for different implementations of

behaviour. This promotes the DIP by allowing higher-level modules to depend on abstractions ('ChitCard'), rather than concrete implementations ('AnimalChitCard' and 'DragonPirateChitCard'), reducing dependencies and increasing flexibility.

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

The relationship between the 'Game' and 'Board' classes is a composition relationship instead of an aggregation relationship because the child object 'Board' is part of the parent object 'Game', and the child object's existence is dependent on the parent object. In this case, the 'Game' class contains a 'Board' object, and without a 'Board', a 'Game' cannot be played successfully. On the other hand, aggregation implies a weaker relationship where the child can exist independently of the object which would not make sense in this case if the 'Game' object is broken, the 'Board' object could still exist. Therefore, in the context of a 'Game' and 'Board', the composition relationship is more appropriate because the 'Board' is an essential part of the 'Game'. This relationship implies a stronger connection and suggests that the 'Game' manages the lifecycle of the 'Board'.

In addition, there is an association relationship between the 'Dragon' and 'CaveTile' classes. Each 'Dragon' has its own identifier, which is used to check if the dragon has reached its corresponding cave tile. If a dragon reaches its cave tile, it triggers the win condition, ending the game immediately. This association highlights the dependency between dragons and their respective cave tiles in achieving the game's objective.

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

In my design, the decision to use inheritance was made to facilitate the creation of different types of 'ChitCard' objects while ensuring code reusability and maintainability. By defining an abstract class 'ChitCard' with common attributes and behaviours shared among all chit cards, such as 'chit_steps', 'chit_quantity', 'flipped', 'flip()', and the abstract method 'perform_action(Dragon)', we establish a clear and structured hierarchy for chit cards.

The use of inheritance allows us to create specific types of chit cards, such as 'DragonPirateChitCard' and 'AnimalChitCard', that inherit the common attributes and behaviours from the 'ChitCard' class. This promotes a modular and scalable design, making it easier to add new types of chit cards in the future without duplicating code.

Furthermore, by defining 'perform_action(Dragon)' as an abstract method in the 'ChitCard' class, we ensure that each subclass must implement its own version of this method, enforcing the specific actions associated with each type of chit card. This not

only adds flexibility to the design but also helps in maintaining consistency and clarity in the codebase.

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

The cardinality of exactly 1 from the ‘Game’ class to the ‘Board’ class indicates a one-to-one relationship between these classes, meaning that each game can only have one board. This relationship is crucial in the context of the Fiery Dragons game, as the board is an essential component without which the game cannot be played.

Using a cardinality of 0…1 (zero to one) would imply that a game can still be played without a board, which contradicts the fundamental requirement of the Fiery Dragons game. Therefore, a cardinality of exactly 1 ensures that each game instance is associated with one and only one board, reflecting the game's design and ensuring its integrity.

Furthermore, the cardinality of exactly 28 from the ‘Board’ class to the ‘Tile’ class indicates that each ‘Board’ has a total of 28 tiles. Among these tiles, 24 are normal tiles, and 4 are cave tiles. The use of this cardinality allows for flexibility in the game design, as it can be easily adjusted if the board size needs to be changed to accommodate more dragons or additional gameplay elements. However, for the basic requirements of the game, the cardinality of exactly 28 is sufficient to represent the fixed number of tiles on the board.

Design Patterns

In my design, I applied the **Observer** pattern to handle events such as when the player picks a chit card and when a dragon returns to its own cave, triggering the win condition. For example, when a player picks a chit card, the ‘Board’ class automatically checks if the animal on the chit card matches the animal on the tile that the player is standing on. If so, it will perform the action associated with the chit card by calling the ‘perform_action(Dragon)’ method. Similarly, when a dragon returns to its own cave, the Dragon class can act as the subject and notify the Game class, which can then update the state of the game, triggering the win condition and ending the game if necessary.

In addition, I have applied the **Singleton** pattern to ensure that there is only one instance of the ‘Game’ class so that there is only one central point of control for the game’s logic and flow. The ‘Game’ class uses a class variable ‘_instance’ to store the singleton instance of the class. The ‘__new__’ method ensures that only one instance of the ‘Game’ class is created, and subsequent calls to ‘Game()’ return the existing instance.