FIT3077 Design Rationale Rohan Sivam 32880316
Design Rationales (explaining the ‘Whys’)
○ 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?
○ Explain two key relationships in your class diagram, for example, why is something an aggregation not a composition?
○ Explain your decisions around inheritance, why did you decide to use (or not use) it? Why is your decision justified in your design?
○ Explain how you arrived at two sets of cardinalities, for example, why 0..1 and why not 1…2?

KEY CLASS - TILE

The abstract class Tile is an abstract class that allows different functionality for the place_player_on_tile methods within the class. The tile class adds functionality to each of the tiles that the GameBoard consists of and is the parent class of the classes NonCaveTile and CaveTile.

(Single Responsibility)
The reason I created the Tile class instead of implementing the functionality inside of the GameBoard class is because implementing the functionality within the GameBoard class would violate the principle of the single responsibility. Having the tile functionality which is assigning the player’s animal and checking if the chit card matches the tile currently occupied is a different responsibility from the GameBoard. The GameBoard is just meant to move the components of the game board around , not to handle the interactions between the pieces. To avoid this, we created a new class Tile to handle the interactions between the Dragon Piece and the Tile as well as the Chit Cards and the Tile.

(Open-Closed Principle)
The first reason I used an abstract class is because of the Open-Closed principle. By implementing the abstract method, this allows us to add additional tile classes with additional functionality later in the game without modifying the tile class. We also don’t modify the place_player_on_tile method’s behaviour as they differentiate between the NonCaveTile class and CaveTile class. If I didn’t use the abstract class and instead just had tiles under the base class Tile, the method would need to rewrite the class to implement new functionality if we implemented new features for the tile class. This would violate the Open-Closed Principle.

(Liskov Substitution)
Both NonCaveTile and CaveTile inherit from the Tile class as they both implement all the methods that the Tile class implements. This means we do not violate the Liskov Substitution principle which is that every subclass should be substitutable for their base or parent class. Any place in my implementation that would have a tile class, the CaveTile and NonCaveTile can be substituted as they have all the same methods and attributes. The reason I didn’t have

all GameBoard pieces under one class is this reason. If Tiles, ChitCards and Dragons, all inherit from the parent class GameBoardPiece, they would violate the principle of Liskov substitution as Tiles cannot be substituted for Chit Cards or Dragons and Vice versa.

The design pattern used for this class is the strategy pattern. This allows my Tile class to be decoupled from the GameBoard and the game implementation. By creating this abstract class, I can create child classes that inherit from this class all implementing the same methods and all being interchangeable for each other. This would allow me to modify the functionality of the tiles, increase the number of tiles on the game board as well as which tiles go where on the game board.

KEY CLASS - CHITCARD

The key class ChitCard is an abstract class that represents the chit card in the fiery dragons game. ChitCards are cards that when flipped over allow the player to move forward or back depending on which chit card is flipped. If the chit card doesn't match, the player's turn ends.

(Single Responsibility)
The reason I created the ChitCard class instead of having a singular GamePiece class or implementing it in the game board is to follow the principle of Single Responsibility. If I were to implement the ChitCard class instead as a GamePiece class with the Tiles and Dragons, this would mean that the GamePiece class has more than one responsibility, by having ChitCard as its own class, it's only responsibility is to determine how many spaces a player moves and in which direction, or if they move at all.

(Open Closed)
The next reason I created an abstract class would be to follow the Open Closed Principle. Each chit card has a different behaviour. The AnimalChitCards need to check if the player's current tile matches the Chit Card while the DragonPirateChitCard moves the player back without checking if the player's tile matches or not. As these two ChitCards have different behaviours, implementing the different behaviours would mean modifying the ChitCard class. If i were to add more ChitCards with different behaviours, having them all under one class would violate the Open Closed principle. By instead having a ChitCard class that is inherited by different type of ChitCards, the ChitCard class is not modified and follows the Open Closed principle as it allows for different extensions.

(Liskov Substitution)
As we may want to change the number of AnimalChitCards or DragonPirateChitCards, or even add additional Chit Cards with different functionalities, by creating the abstract parent class of ChitCard, we follow the principle of Liskov Substitution. If I implemented the ChitCard class under a GamePiece class, ChitCards would not be able to execute the methods that other game pieces such as game tiles follow. This allows for any ChitCard to be implemented in the place of the other ChitCards, if I were to implement a ChitCard that for

example would allow two players to switch places, this would follow the Liskov Substitution principle. Any child chit card be substituted in place of the parent Chit Card class

Another design pattern that is used for this class would be the observer pattern. The game board class that checks for when a chit card is flipped and then proceeds to carry out the actions that result from the chit card is flipped. The chit card and the game engine are loosely coupled. If an animal chit card is flipped, the game engine listening will then proceed to check if the tile the player that flipped the chit card is on matches the flipped chit cards animal. If it does move the animal, if not end the player's turn. The respective interaction is carried out for DragonPirateChitCard if a dragon pirate chit card is flipped

Finally, one last design pattern I will use is the singleton design pattern. The reason is because we do not want to instantiate more than one FieryDragons instance at a time. This is because we only want to instantiate the game once when running it. This prevents resource sharing between multiple game instances at one time. As well as this, it allows for lazy instantiation as we do not want to initialise the game unless we request to start the game. This also allows the chit cards and the game board as well as the dragons to interact with each other without instantiating new instances.

KEY RELATIONSHIPS

Why is the relationship between GameBoard and ChitCards an association instead of an aggregation or composition. Firstly, the reason why the relationship between GameBoard and ChitCards is not a composition relationship is because the Chit Cards can exist without the Game Board. The Game Board is made up of the Chit Cards, the Players and the Tiles, However, the ChitCard objects exist on their own and can be used for different game board objects.

The reason the relationship is not an aggregation is again because of the fact that the chit cards do not belong to the game board, the chit cards can be used for multiple games and different game boards, different chit cards can also be used for the same game board. As the game board does not have specific chit cards assigned to it, it is not an aggregation relationship and is thus a Association relationship

Why is the relationship between FieryDragons and GameBoard an association instead of an aggregation or composition? Firstly, the reason why the relationship between FieryDragons and GameBoard is not a composition is because the GameBoard can exist on its own without the existence of FieryDragons as a game. We only play FieryDragons on the game board, the game board does not only exist to play Fiery Dragons on it.

The reason the relationship is not an aggregation is because of the fact that this GameBoard does not belong to this FieryDragons game exclusively, different game Boards can be used to play Fiery Dragons and different Games can be played on the Game Board. As the game board does not belong to the FieryDragons game it is played on exclusively, thus it is only an Association relationship.

INHERITANCE

The way I used Inheritance was for my abstract classes Tile and ChitCards. This is firstly because I wanted to follow the Open Closed principle, this meant that my classes must be closed for modification but open for extension. As the Cave Tiles and Non Cave Tiles have different behaviours, by using a parent class that implemented the main functionality of a Tile but implemented the methods differently in each of the Child classes, I was able to follow this principle. This allowed me to implement the different behaviours of the place player on tile method which placed a dragon token on the child. By using inheritance and abstraction, I made it so that normal tiles did not check if the player number matched the tile to move the player to the tile while the cave tile does. If i didn't use inheritance, I would have to implement the functionality of both tiles within one method. If I were to add more tiles with different functionality, this would violate the open closed principle. This way, I can add more Tile types with different features and functionalities without modifying any of the previously created classes. This works the same for my Chit Card classes. This is also useful in my current game as Dragon Pirate Chit Cards can only have up to 2 dragon pirates on them while the Animal Chit card class can have up to 3. By using inheritance, when creating my chit cards, I can instantiate the Chit Cards a lot easier.

CARDINALITIES DRAGON 2..4 FIERYDRAGONS 1

The way I came about the decision of having 2..4 for the Dragons that play Fiery Dragons is because of the fact that there needs to be a minimum of 2 players to play Fiery Dragons and as of the base game rules, there can only be a maximum of 4 players. This is represented by the 2..4 cardinality, as there are 2..4 players per game , that is why there is a 1 for Fiery Dragons as there can only be one instance of FieryDragons game at a time that the players partake in.

CARDINALITIES CHITCARD 16 GAMEBOARD 1

The way I came about having the decision of having 16 as the cardinality for the ChitCards and not any other number is because in the game rules, it states that there should be 16 chit cards on the game board. The 16 chit cards consist of 12 animal chit cards and 4 dragon pirate chit cards. One game board can only have 16 chit cards which is why there is 1 for game board and 16 for chit cards. The gameboard should not have less than 16 which is why it is just 16 and not 0..16. The game board should also not have more than 16 which is why it is not 16..*. Only 16 chit cards are required which is why it is just 16.