FIT3077 System Architecture and Design

SPRINT TWO DOCUMENTATION

A summary of how the key game functionalities of

(i) set up the initial game board (including randomised positioning for dragon cards)

The initial game board is set up in the FieryDragons class by inputting a configuration of dragon cards and configuration of volcano cards.

To create the DragonCards for the game, the playGame function takes this card configuration, which includes a combination of Animal classes to indicate the cards' types as well as the number of animals on each card. From there, the FieryDragons class creates the corresponding number of AnimalCard classes, each with an animal attribute, and number of animals on that card. In the same way, it will create the corresponding number of DragonPirateCards with the given number of dragon pirates on this card.

To create the players and the GameBoard itself, the playGame function uses the given board configuration to determine the number of players in the game, through the number of caves given. Thus, it firstly creates Caves. For each cave, it then adds this cave as the home and location attribute for a new Player. From there, the playGame function creates Square objects following the configuration's selection of animals and number of animals, linking the previous Square and next Square to this new Square object. This creates a linked list of Square objects for the program to traverse throughout the game. For Squares connected to caves, the cave attribute is set to the linked Cave. This is only returned by the getNextLocation function if the number of moves remaining is 1.

**could have used an adapter for the cave but it wasn't being used elsewhere so just implemented the methods directly*
**mention something to do with both of these being extendible, like the pirate class can be extended by adding category classes just like the animal classes and use them in the same way*

(ii) flipping of dragon ("chit") cards;

The dragon cards are flipped with the pickCard method within the CardsController. This displays all dragon cards on the screen in a grid, along with numbers corresponding to each card for players to select. Once a player has selected a card, the selected card is displayed with its animal and number of animals on the card, and is noted as being turned over through its setUnavailable method. This is reset at the end of each round using the resetCards method in the cards controller.

(iii) movement of dragon tokens based on their current position as well as the last flipped dragon 1 card

The flipped dragon card contains a moves function which, with the input of the player's current location, determines if the player can move with the DragonCard (this is set to true for the DragonPirateCard and is dependent on the location's animal for AnimalCards). If this method returns an integer other than 0, then the getNext method is called with an input of the player's current location. This calls the location's getNextLocation or getPreviousLocation method depending on whether it is a DragonPirateCard or AnimalCard, and uses an input of moves. This is the value of how

many animals or pirates are on the DragonCard, and is used to return the next location. If the number of moves is 0, the function returns the current location. If the number of moves is above 1, the function will only return the nextLocation, but if it is exactly 1, the function will return the linked Cave followed by the Square in an array.

This will be checked by the player's setLocation function. This takes an input of an array of Locations and checks if the player can move to either of the locations. As the Cave class's canMoveTo function will only return true if the number of moves matches the number of totalSquares in FieryDragons, and the Square class will always return true, the Player sets their location to the first Location that they can move to, which will always be the Square, unless they have moved all the way around the board.

Before a player moves to a destination, they set their previous location's occupied status to false using the setUnoccupied method. They then move to the next destination and set that location's occupied status to true. If another player is already sitting at the player's destination square, the canMoveTo function will return false and thus prevent the player from moving to this position.

If all of the cards have been previously selected, the pickCard function from the CardsController will return None, as no cards will be available to pick from. This will result in the game progressing to the next player.

(iv) change of turn to the next player

Once the player selects a DragonCard, the player's location is passed to the moves function of the DragonCard. For DragonPirateCards, this is always set to the number of moves from the moves attribute, but for animal cards, this checks if the card's animal matches the location through the Location's matchAnimal function. If the matchAnimal function returns true, the function correctly returns the number of animals on the card, the player moves and can have another turn, but if not, the function returns 0 and the game moves to the next player's turn.

(v) winning the game.

The game checks if a player's current location matches the home location with the function checkWin. This function checks if the current location matches the home attribute location, and if so, the player has won the game.

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?

The AnimalCard class was created to specifically handle whether a Player was able to move using their selected AnimalCard or not, as well as where the Player would move to. Although this could have been possible with a method, it would have resulted in a confusing, complex design that would be difficult to extend and adjust. The AnimalCard also individually stores information about whether it has been previously selected by a player, which could have been stored in an array for all cards, but again, creates additional complexities and makes it difficult to manage, particularly with trying to set

specific cards as being selected. The decision to make an AnimalCard class was intuitive as the AnimalCard represented a physical object in the game that had properties of animal types and number of animals. By separating this data by card into individual DragonCard classes, this simplified the code, made it easier to manage and will allow for further extensions in the future with the creation of more cards or more types of cards.

The Square class was created to represent a specific location on a volcano card. This was created specifically to manage how players moved around the board based on available moves. Whilst this also could have been represented as an array, this would have added additional complexities, particularly with linking array positions to caves. In addition to storing previous and following locations, the Square class is also able to store information such as its Animal classification, and whether it is occupied or not, which is useful in verifying whether a player can move to or from a location.

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

The CardsController and PlayerIterator relationships with the FieryDragons class were classified as compositions, as the FieryDragons game requires both a CardController and PlayerIterator to run, and neither the CardController or PlayerIterator has a use without the FieryDragons class.

The relationship between a player and a location however, was classified as an association, as the location was independent of the player, its life existing beyond being contained as an attribute for the player as the player moved from one location to the other.

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 used in my design for the DragonCard classes, Location classes and Animal classes.

For the DragonCard classes, both DragonPirateCard and AnimalCard inherited from the DragonCard abstract class. This was due to their similar properties of representing a card a player could select and allow the player to move either forwards or backwards. By ensuring that both of these child classes inherited from DragonCard, this allowed the playGame function to simply ask each class with the same methods, to find the next location and get the card's number of moves to pass to the player, regardless of whether the card was a DragonPirate or Animal card. It also allowed for the similar functionality of the cards to be similarly marked as available and unavailable. These cards were split into separate child classes due to their differences in functionality. Whilst the AnimalCard class needs to check if the player's current location matches in terms of animal classification, the DragonPirate card does not.

Inheritance was also used in the Location classes. In creating a class for a Cave and for a Square, there were similarities in their functionality, particularly with their need to be able to find the next linked location, and verify if a player can move to that position. Whilst there were differences in the child class properties, such as some Squares requiring Cave attributes, attributes such as the animal classification and following

Square were the same across the two classes. There were also differences in the methods needed for each class, but given that the two classes needed to be treated in the same way for the purposes of moving players across the board, the classes needed to be combined in some sort of way. Although an adaptor could have been used for the Cave class in addition to the inheritance, as the Cave class is not used anywhere else in the application, implementing the methods within the Cave class reduced the need for unnecessary class hierarchy. This allowed for all the same methods to be called on both the Caves and Squares, regardless of their individual type. As each class needed to respond differently in each method, this allowed for Caves to return appropriate values for movement based methods like canMoveTo and Squares to properly validate previous and next Location results and check if the location is occupied.

Inheritance was used in the Animal classes as opposed to enumeration, to allow for each Animal child class to have a display character to be used in the display of the game. As each Animal had this similar property of having a display character, and was used in identical ways by both AnimalCards and Locations, it was important to classify these under the same abstract class of Animal. This allows for the game to be more extendable, as to add further types of animals, more child classes could be easily added.

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

FieryDragons to PlayerIterator needed a 1 to 1 cardinality as the game only requires 1 PlayerIterator to run at a time

FieryDragons to CardsController needed a 1 to 1 cardinality as the game only requires 1 CardsController to run at a time

CardsController to DragonCard required a 1 to 1..* cardinality as it requires at least 1 DragonCard to run the game, and has an upper limit of as many DragonCards as given by the configuration. The DragonCards are all stored in 1 CardsController.

PlayerIterator to Player required a 1 to 2..* cardinality as it requires at least 2 Players to run the game, and has an upper limit of as many Players as given by the configuration. The Players are all managed by 1 PlayerIterator for the whole game.

FieryDragons to Cave required a 1 to 1..* cardinality as 1 game of FieryDragons stores as many Caves as there are players, which has a minimum of 2 and an upper limit of infinity.

Player to Cave required a 1 to 1 cardinality as each Player is allocated to 1 unique Cave.

Player to Location required a 1 to 1 cardinality as each Player can only land on 1 location at a time.

AnimalCard to Animal required a 1 to 1 cardinality as each AnimalCard is categorised to be a single type of Animal.

Location to Animal required a 1 to 1 cardinality as each Location is categorised to be a single type of Animal.

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.

Facades were used in this design, particularly in the creation of the CardController and PlayerIterator. These were used as opposed to arrays or similar data structures to allow for the CardController to use an observer pattern (as described in a later section) and for both classes to decouple the FieryDragons client from the subsystems of players and cards. This helped simplify the FieryDragons class and adhere more closely to the single responsibility principle of OOP design, shifting the responsibilities of managing the players and cards to separate classes. The facade pattern was implemented by considering the FieryDragons class as the client using the facade of the CardController, which manages the subsystem of the individual DragonCards. Similarly, the FieryDragons class, as the client, uses the facade of the PlayerIterator, which manages the subsystem of individual Players. These design patterns allow the FieryDragons class to manage less of the Player and DragonCard interactions and add another level to the system structure. Whilst the game board could also have similarly been separated into a separate class, this would have resulted in a tightly coupled group of classes due to the interactions required between players, cards and the locations, making the system much more complex and more difficult to extend in the future.

Singletons were also used in this design, particularly for the Animal classes. These were implemented as singletons with a private default constructor and static creation method for each of the Animal child class types. This allows for the Animal classes to be easily comparable, linking all DragonCards and Location with the same Animal as their type together, without the need for type checking (as would be needed if the Animal classes were simply added to these classes as separate instances) or enumeration. The Animal classes were implemented as classes to allow for storage of additional Animal data, such as the display character to be used in the application, which couldn't have been included with the use of enumeration.

An observer pattern was used for this design, particularly to be able to notify DragonCards that they should be reset as being "available" for selection after each player's turn. This was done to easily be able to change the state of each DragonCard, whilst adhering to the Open/Closed Principle. This will allow for further child DragonCard classes to be created without needing to change the implementation of the CardController class. This pattern was implemented by considering the FieryDragons class as the client, the CardController class as the Publisher, and the individual DragonCards as the subscribers. As the game is played, selected DragonCards are marked as being unavailable through their setUnavailable method. This will allow for players to only select available, unselected DragonCards for the remainder of their turn. The FieryDragons class then calls the CardController's resetCards method each time at the start of each player's turn to reset each of the DragonCards' availabilities to allow the next player to select any of the cards. This is done by the resetCards method calling each one of the DragonCards' own reset methods, notifying them of the round reset for them to update their availabilities appropriately.

Although the AnimalCards and Squares could have been separated into individual animal card classes, with a factory method used to create each of the animal cards from the configuration within the FieryDragons class, this would have added additional

complexities that do not add much extendability to the architecture. Whilst this would be extendable and allow for each AnimalCard to have different functionalities in future implementations, given that each AnimalCard type still has the same functionality in the current game, it was decided that it was more likely for additional AnimalCard types to be created with identical functions to the current AnimalCard class. Thus, comparing the work required to implement additional Animal types, it would require a lot more work to create additional AnimalCard classes as well as corresponding Square classes, than it would be to simply create an additional Animal class and set it as an attribute for an AnimalCard or Square as is in the chosen implementation. Thus, it was not feasible to create a factory method here, given that there were only two different classes to be created.