FIT3077: Software Engineering: Architecture and design

Semester 1, 2024

Sprint 2 Design Rationale

Group No: MA_Wednesday_04pm_Team690

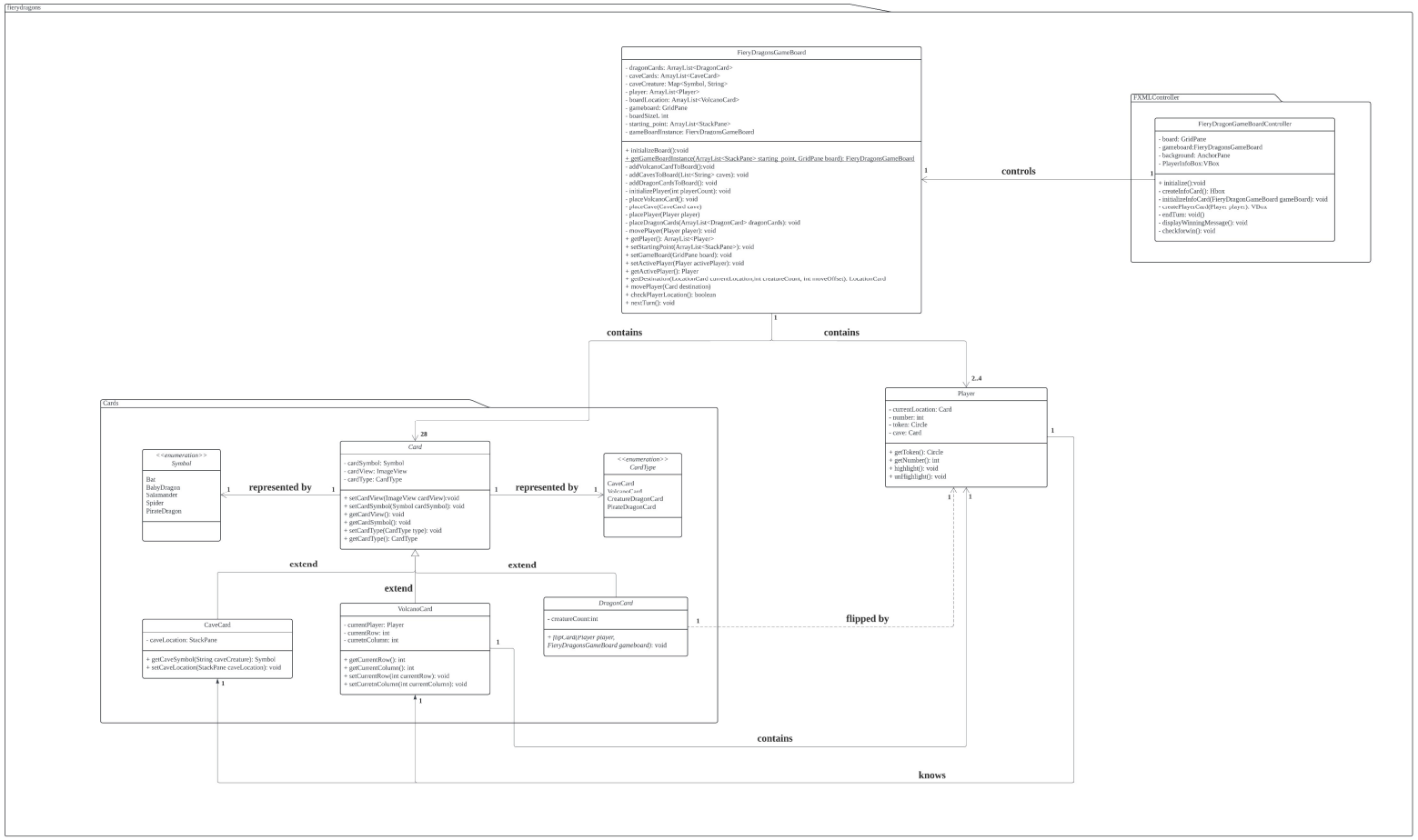
Prepared by:

Tong Jet Kit 32632177 jton0028@student.monash.edu

Contents

UML Class Diagram

Lucid Chart Link: https://lucid.app/lucidchart/5f53778e-44a8-43c5-bacf-0101f3127e71/edit?viewport_loc=-3037%2C-1397%2C8000%2C3700%2C0_0&invitationId=inv_4e782b54-80fa-4066-83c8-077067e3a230



Design Rationale

In my design for this game include several key classes such as Player class and Card class. The Player class is a class that represents the players that will be playing the Fiery Dragon Game. In this game, the system needs to know the location of the player and the player can flip a Dragon Card during its' turn. Thus, rather than putting this information into the gameboard, we create the Player class to be responsible for the player's information and behaviours. Moreover, this will obey the Single Responsibility Principles as we prevent the gameboard to be a God class where it handles everything. This is because we delegate the responsibility of flipping the dragon card to the player rather than the gameboard instead. Therefore, to adhere to the SOLID principles we need to create the Player Class. Furthermore, the Card class is also an essential class that needs to be created. This is because this game revolves around cards. Therefore, there is a need to create a card class. This Card class will represent all the cards of the game, the Volcano Card, Dragon Card and Cave Card. The Card class is an abstract class in this scenario because all the cards share a lot of common attributes and behaviours. This helps reduce code duplication which will be hard to maintain. Furthermore, this allows for a higher level of abstraction, focusing on a design rather than specific implementation details. This is because an abstract class can represent a concept that is not meant to be instantiated on its own but provides a base for derived classes. Therefore, we create the Card abstract class to promote flexibility and code reusability by acting as a base class for all the concrete Card classes like Cave Card and Volcano Card.

In the UML design, there is a dependency relationship between the Dragon Card and Player Card. This is because a player will flip a dragon card during their turn. It is a dependency relationship as the dragon card depends on the player to execute its' responsibilities after being flipped. It is not an association relationship as the dragon card does not need to store the information of who flipped itself but rather it just needs the player's information at the time it is flipped. This will then reduce coupling between the Dragon Card class and the Player Class. Other than that, there is an association relationship between the Player class and the Cave class as well. This is because the Player should know which cave it comes from to win the game. Therefore, the Player must store the information of its' corresponding Cave Card so that when playing the game, the Player would know when it has won the game by returning back to its' starting cave after roaming around the whole board. So, the Player class would have an association relationship with the Cave class.

Additionally, I have created some generalization relationship among the classes representing the cards in the board. I utilized inheritance as it effectively reduces code duplication by allowing codes to be shared among classes. This is because inheritance relationship allows the child class to inherit all the methods and attributes of the parent class. This will then reduce the number of repeated code lines and will improve code maintainability as the methods are centralized which will reduce the risk of inconsistencies. In addition, it also helps to adhere to the Do Not Repeat Yourself Principle. For example, the Volcano Card, Cave Card and Dragon Card class will inherit the Card class. This is because all these classes are cards that share similar attributes like the numbers and type of creatures

on it. Thus, it is crucial to have them inherit from the abstract Card class. In conclusion, I relied on inheritance to promote code reusability and facilitate polymorphism. Furthermore, inheritance encourage flexibility since it allows customized implementation for the behaviour of the child class which makes modification to be better as it does not impact existing codes. Finally, inheritance also aids in promoting extensibility, making it easier to integrate a new dragon card in the future. Thanks to inheritance, the system is designed to accommodate such extensions with minimal disruption.

The number of players for this game is 2 to 4 players. Therefore, the multiplicity between the FieryDragonGameBoard class and the Player class is 1 to 2 to 4. At the Player class side, it is not 1..4 or 0..4 because this game cannot be played by 1 player only and having 0 players makes no sense. Hence, the multiplicity relationship of 1 gameboard has 2 to 4 players is correct. Other than that, between the Dragon Card class and the Player class there is a dependency relationship between them as the dragon card will be flipped by a player. The multiplicity on the relationship is 1 to 1. This is because a card can only be flipped by a player and a player can only flip one card. Thus, it is logical that the multiplicity will be 1 to 1.

Additionally, for the game system in this sprint, most of the initialization of the game cards are hardcoded, this is due to the fact that number and type of cards in the current game system is well-defined and fixed. Thus, it is not necessary to create methods for extensibility. However, it is noted by me that the game system might change in the future, thus I have already come up with strategies to deal with it. I believe this design decision is good for now since it follows well for the agile methodology where changes are expected, and we should not plan for long term goals but short-term goals.

During implementation, while the initial design blueprint provided a foundational framework, I have identified specific areas where adjustments were necessary. Firstly, I created a new class call VolcanoCardGroup. This class is used to represent the tiles that make up the 3 volcano cards. I decided to create this class since the volcano cards for this game is not a single card, so I have to create a class that groups them up. Although this will make the class to be like a data holder, but I believe my decision in making this class is wise since it is better to let the VolcanoCard class to represent a single tile as this approach simplifies the design, focusing the VolcanoCard class on managing a single tile's core attributes and behaviors. By isolating this functionality, we promote clarity, reduce the risk of side effects, and make the codebase more maintainable. Moreover, this encourages modularity as it makes it easier to replace the tiles with other cards that maybe created in future sprints without affecting the structure of the system. Moreover, I have also created a new abstract class call LocationCard that the CaveCard and VolcanoCard class extends from. My decision for creating this class is to allow the gameboard to get the destination card through the player’s current condition. Without this abstract class, this will make the player need to take note of its’ current location if it’s a cave card or a volcano card and this will make the player to be coupled by both classes. Thus, it is wise to create a LocationCard abstract class to abstractive the player’s current location.

Furthermore, other than creating new classes, I also assigned some new responsibilities to other classes and shifted certain responsibilities from one class to another. For instance, I added 2 methods for the Player class. The methods are the method to create their own tokens and the method to move itself across the board. Originally, the movement method was delegated to the gameboard function. However, after brainstorming I have decided to delegate this responsibility to the Player class instead. By letting the Player class handling its own movement, this will adhere to the Single Responsibility Principle, a core tenet of object-oriented programming. It ensures that the Player class has one clear responsibility: to manage player-specific behaviour, Conversely, the FieryDragonsGameBoard class can focus on board-specific operations like managing tiles, checking game rules, or interacting with multiple players rather than being a God class that manages all operations of the game.

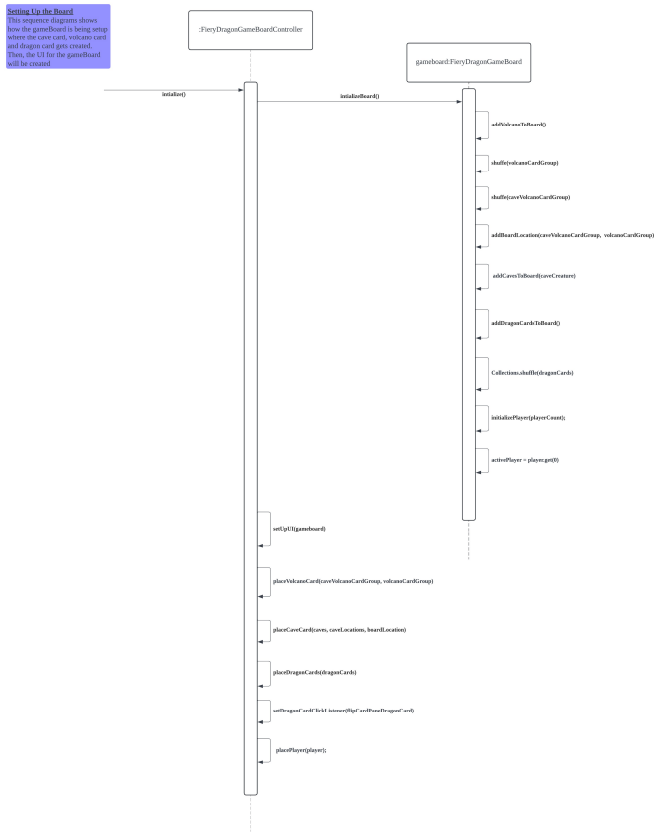
Additionally, I also decided to delegate the task of creating the graphics for the gameboard and the onclick method to flip to the controller class rather than it being the gameboard's responsibility. This is so that I do not make the gameboard to be a God class that handles responsibility that is out of its' scope. Thus, it will adhere to the Single Responsibility Principle since the controller class is a class that should handle anything relating to the user interface.

Summary of Key Functionality

I would like to note that for this sprint, my tasks is to set up the initial game board and flipping of dragon cards.

UML Sequence Diagram: Setting Up the Gameboard

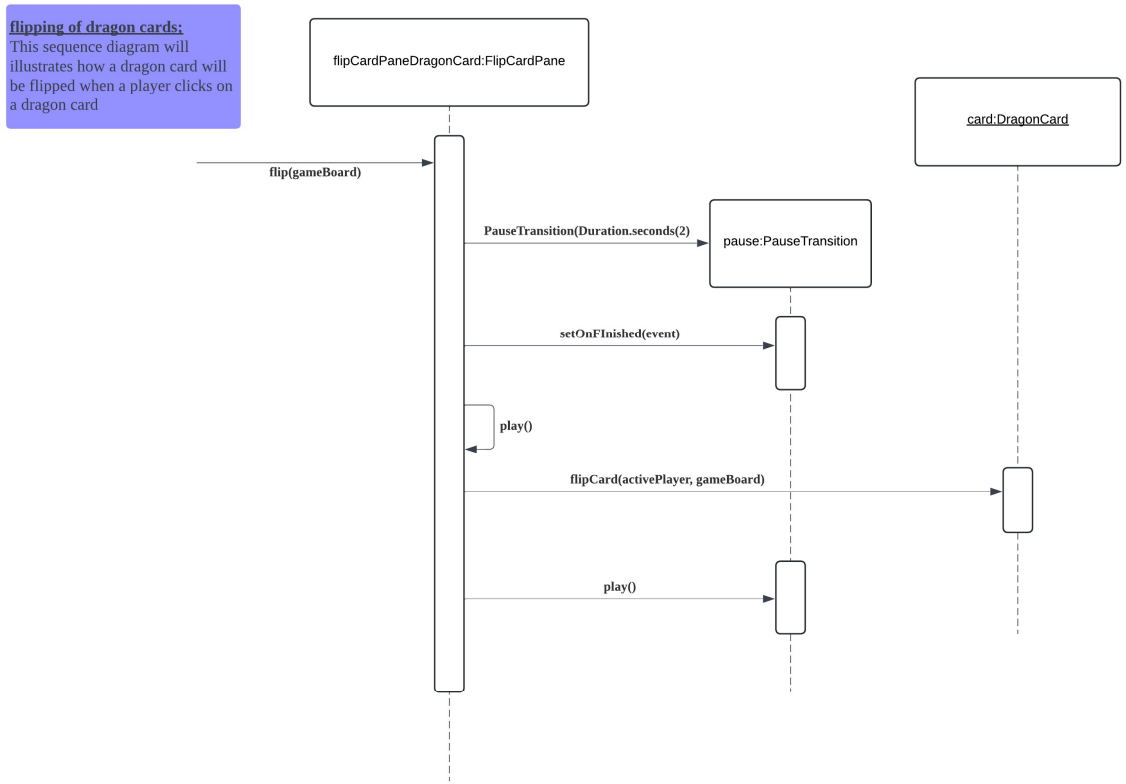
Lucid Chart Link: https://lucid.app/lucidchart/5f53778e-44a8-43c5-bacf-0101f3127e71/edit?viewport_loc=-926%2C-179%2C1707%2C789%2CI_qwTRvzp.~g&invitationId=inv_4e782b54-80fa-4066-83c8-077067e3a230



This sequence diagram shows the board setup on how it creates the objects for the game and the user interface. The gameboard controller class will initialize the board by calling the initializeBoard() method. This method will then create the volcano card tiles, the cave cards, the dragon cards and then the players accordingly. Then lastly, the gameboard will note down the active player which will be player 1. Then once the gameboard is initialize, the controller class will then initialize the user interface for the game cards. The controller class will first create the graphic for the volcano card, where the images will be rotate accordingly. Then, it will create the graphics for the cave cards. Next it will add the dragon cards to the center of the board while also adding an onClickListener to each of the dragon cards where when the player clicks on it, it will flip the card and move the player's token. Finally, the controller class will create the graphic for the player's token and set it on the player's respective cave card since the cave card is the starting location for the player.

UML Sequence Diagram: Flipping the Card

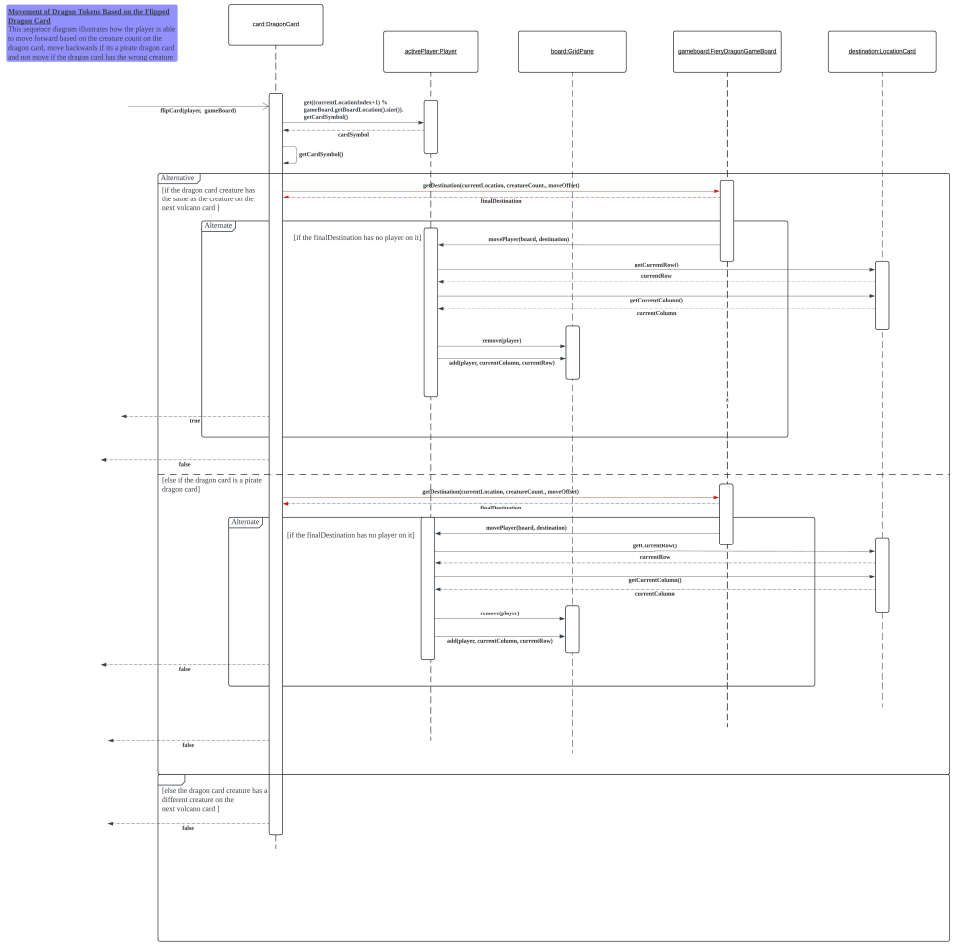
Lucid Chart Link: https://lucid.app/lucidchart/5f53778e-44a8-43c5-bacf-0101f3127e71/edit?viewport_loc=-143%2C-600%2C2303%2C1065%2C4rOw0KE0zKPs&invitationId=inv_4e782b54-80fa-4066-83c8-077067e3a230



This sequence diagrams illustrates what happens when a player wants to flip a card. When the player clicks on the dragon card, it will then call the flip() method. In this method a pause transition will be created so the card will be flipped back face down after waiting for a while. This is done using the setOnFinished() method to trigger the flip animation again. Before flipping back, the flipCard() method is invoked on the clicked card so that it can handle the movement of the player based on the chosen dragon card. After that the card will be flipped back. During these flipping animations, the board will not allow anymore clicking until the current clicked dragon card has been resolved. Only then, another dragon card can be clicked again.

UML Sequence Diagram: Movement of Dragon Tokens based on the Flipped Dragon Card

Lucid Chart Link: https://lucid.app/lucidchart/5f53778e-44a8-43c5-bacf-0101f3127e71/edit?viewport_loc=-1%2C759%2C1707%2C789%2CQYwwzOTkNfOL&invitationId=inv_4e782b54-80fa-4066-83c8-077067e3a230

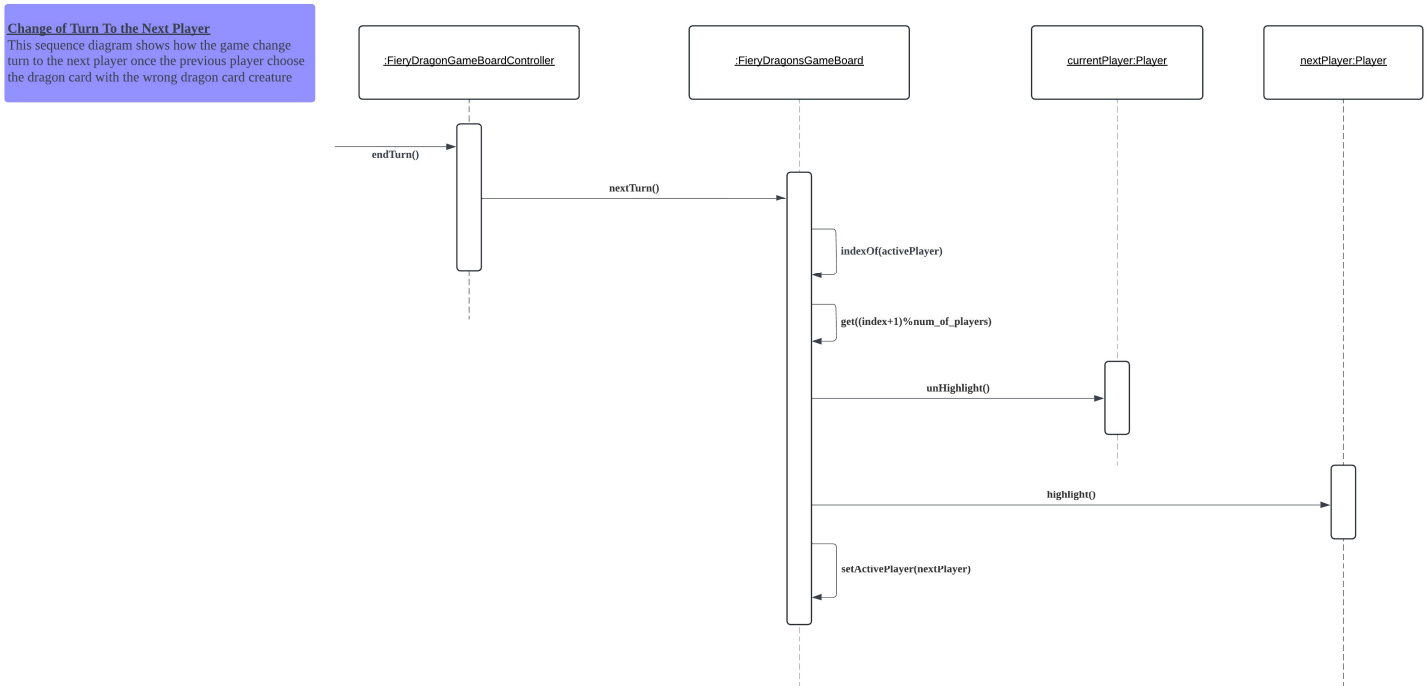


This sequence diagram shows how the movement dragon tokens based on the flipped dragon card based on the flipped dragon card. After flipping it will first get the card symbol for the next volcano card. There are 3 scenarios when flipping the card.

First scenario is that the flipped dragon card has the same creature as the next volcano card. When this happens, the getDestination() method will get the destination volcano card for flipping that dragon card. Then it will check if the destination volcano card has a player on it. If there is a player, then it will not move and return false since the player turn will end; otherwise, the player will move to the destination volcano card. This goes the same way for the second scenario when the player chooses the pirate dragon card but just move backwards. Moreover, after the player move backwards, the player turn will end so it will return false, nonetheless. Lastly for the third scenario is if the flipped dragon card has a different creature than the next volcano card, it will end the player's turn thus, it returns false.

UML Sequence Diagram: Change of Turn to the Next Player

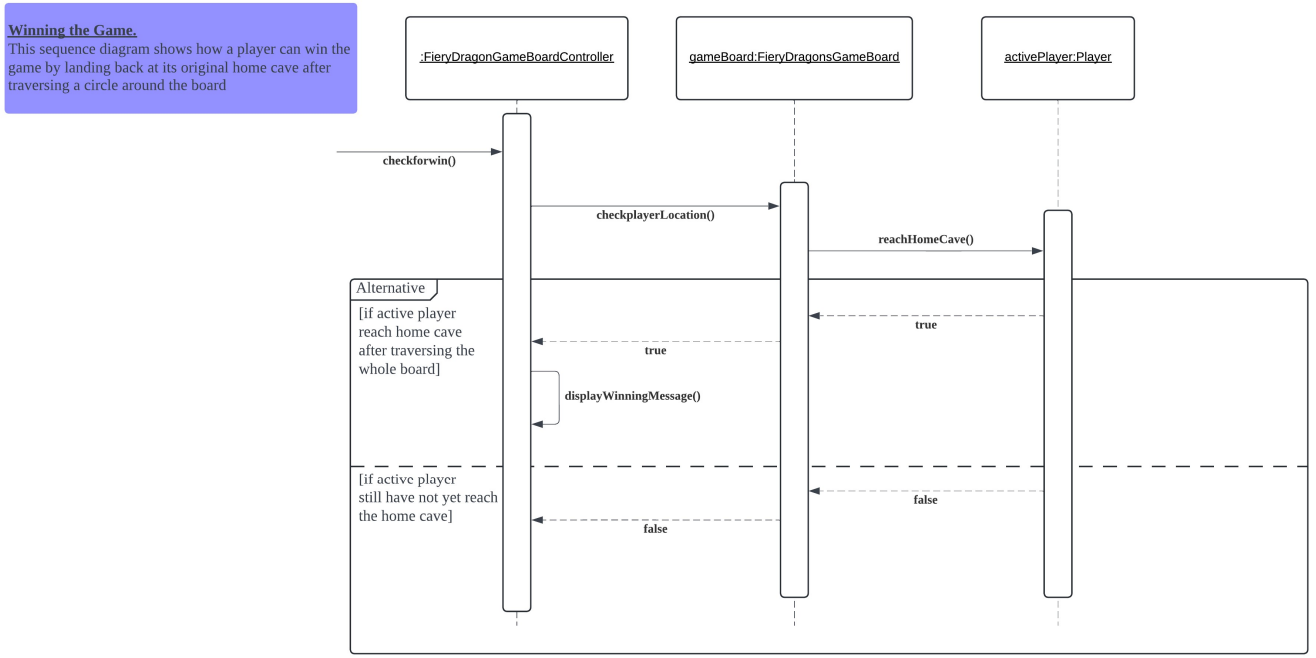
Lucid Chart Link: https://lucid.app/lucidchart/5f53778e-44a8-43c5-bacf-0101f3127e71/edit?viewport_loc=166%2C203%2C1707%2C789%2C.36wEJH2Gmey&invitationId=inv_4e782b54-80fa-4066-83c8-077067e3a230



This sequence diagram shows how the change of turn to the next player once the previous player chose the wrong dragon card. When endTurn() method is called, the gameboard will go get the next player's index through the formula $(activePlayerIndex + 1)\ \%\ num\_of\_players$. Then the current active player will be unhighlighted using the unhighlight() method and then the next player will then be highlighted using the highlight() method. This is to show the change of turn from 1 player to another. Finally, the gameboard will then take note of the next active player.

UML Sequence Diagram: Winning the Game

Lucid Chart Link: https://lucid.app/lucidchart/5f53778e-44a8-43c5-bacf-0101f3127e71/edit?viewport_loc=121%2C-502%2C1707%2C789%2CdO8wA8tbpRVh&invitationId=inv_4e782b54-80fa-4066-83c8-077067e3a230



This sequence diagram demonstrates how the game decides when a player win a game. After each time a player move forward after flipping the right dragon card, the controller class will check if a player has reached its' home cave after traversing the whole board. This is done by calling the checkplayerLocation() in the FieryDragonsGameBoard class. This method will check if the current active player has reached its' home cave card. If they have reach it, then the controller class will invoke the displayWinningMessage() method to announce that the current active player has won the game. Otherwise, if the player hasn't reached their home cave, the game will continue as usual.

Design Patterns

In my design I have used the Template Method Design Pattern. The Template Method Design Pattern defines a skeleton class that outlines some basic implementation that are common for all or some of its' subclass. This skeleton class can be inherited by their subclass to override specific implementation to suit its' needs. I utilize this design pattern as it promotes code reusability, flexibility, and extension by allowing subclasses to implement specific behaviours. By using this design pattern, the common shared implementation will be centralized in one location which is the parent class. This will then help in code maintenance since there will be no code inconsistencies when updating the class. Other than that, by using the template method design pattern, we can then improve the flexibility since the subclass can customize the implementation based on their needs. Lastly, we can extend the game easier with this design pattern since the skeleton class acts as a framework that defines the general attributes and behaviours. Then, we can introduce new subclasses that conforms to the framework structure without modifying the existing code making the overall system extensible.

As an example, the design can be seen between the Card parent class and the DragonCard child class, CaveCard child class and VolcanoCard child class. The Card class is a template for the CaveCard, DragonCard and VolcanoCard classes. I applied the template method pattern design since in the game there are 3 types of cards. These cards have common features, like having a creature depicted on them, but they exhibit unique behaviors. For instance, players can flip the dragon cards, while the volcano and cave cards cannot be flipped. Each card type also has its own set of custom attributes. The template method design pattern is ideal for this situation, as it allows me to define a common structure for these cards while enabling customization for their specific behaviours. This pattern reduces redundant code and fosters flexibility and extensibility through inheritance, making it easier to maintain and extend the codebase.

Additionally, I have also implemented the Singleton Method design pattern for the gameboard object. I chose to implement the Singleton design pattern for creating a single instance of the gameboard object to ensure that the game maintains a consistent state across all its components. The Singleton pattern enforces a single point of access to the gameboard, preventing multiple instances from being created. This approach is crucial in scenarios where the gameboard represents a central resource or shared context that must be consistent throughout the game lifecycle. By using the Singleton pattern, I can avoid potential conflicts or inconsistencies that might arise if multiple instances were inadvertently created, leading to improved stability and easier management of the game's core logic. Additionally, the Singleton pattern provides a clear and centralized way to access the gameboard, simplifying code structure and reducing the likelihood of errors related to object references or initialization. Ultimately, this design choice contributes to a more robust and maintainable game architecture.

Comprehensive Testing

Test for setting up the gameboard.

The test case will test if the user interface shows all elements of the Fiery Dragon game where:

a. Dragon board is set up correctly where there are 4 cave cards place at the side and 24 volcano cards tiles.
b. Dragon cards are arranged inside the ring of volcano card and clearly visible as turned over and undistinguishable.
c. Player tokens are set up in the correct starting position.

Testing results:

PASS

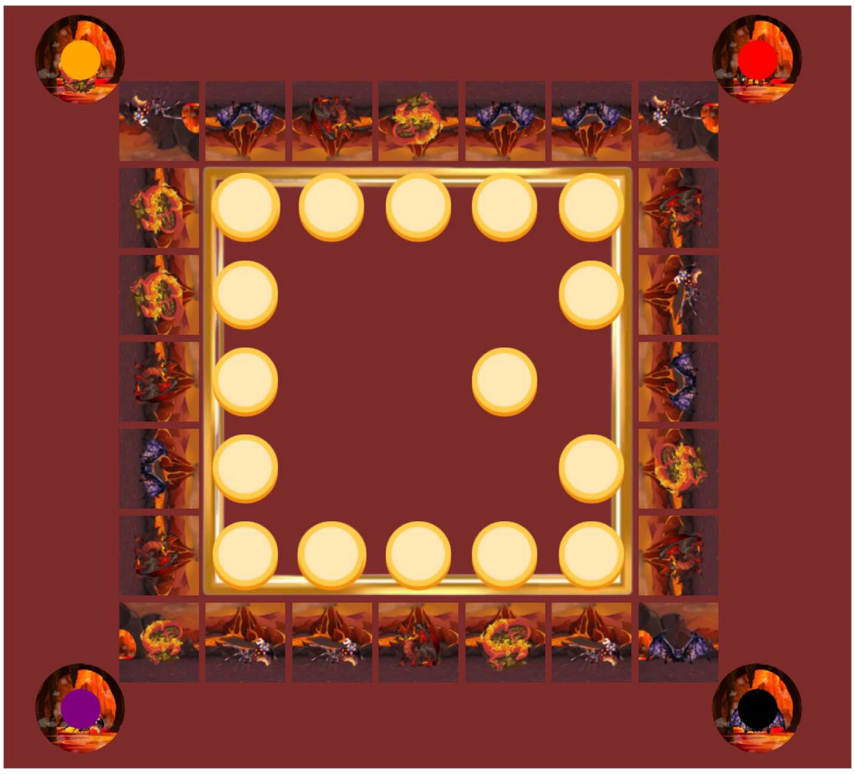


Figure 1: Gameboard Setup

As shown in the picture above, there are 24 volcano card tiles placed neatly to form a square ring, 4 cave cards placed perfectly and there are 16 dragon cards that are arranged neatly in the center of the volcano cards. Lastly, the player tokens are initially set up at their respective starting location which is in the cave cards.

Thus, the test to check if the gameboard is set up correctly has been successful.

Flipping the Card

This test case will test if the dragon cards can be flipped by the player.

Testing result:

PASS

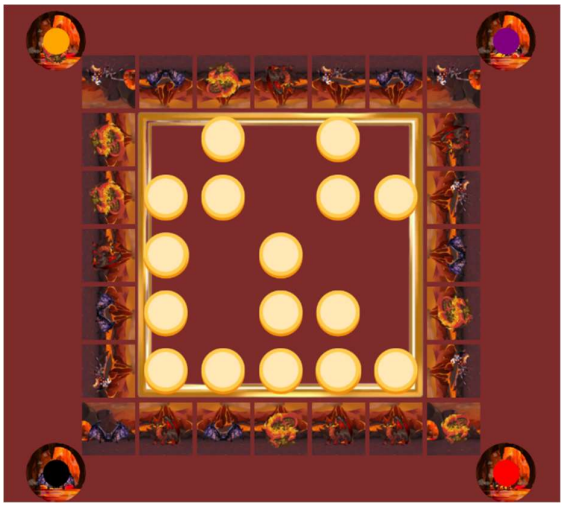


Figure 3: Unflip Dragon Card

Figure 2: Flipped Creature Dragon Card

Figure 4: Flipped Pirate Dragon Card

As shown in the figures above, the Figure 2 represents the gameboard before a player clicks on a dragon card, while Figure 3 shows the gameboard if the flipped dragon card is a creature dragon card and Figure 4 shows the gameboard if the flipped dragon card is a pirate dragon card. These images demonstrate that a creature dragon card or a pirate dragon card can be revealed after a player clicks on a dragon card.

Thus, the test to check a player can flip a card is successful.

Randomized Dragon Card

This test case will test if the dragon cards have randomised positioning.

Testing result:

PASS

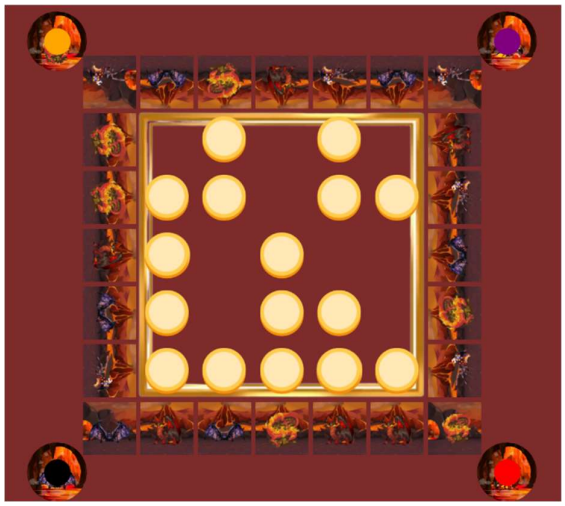


Figure 5: Game State 1

Figure 6: Flipped Dragon Card for Game State 1

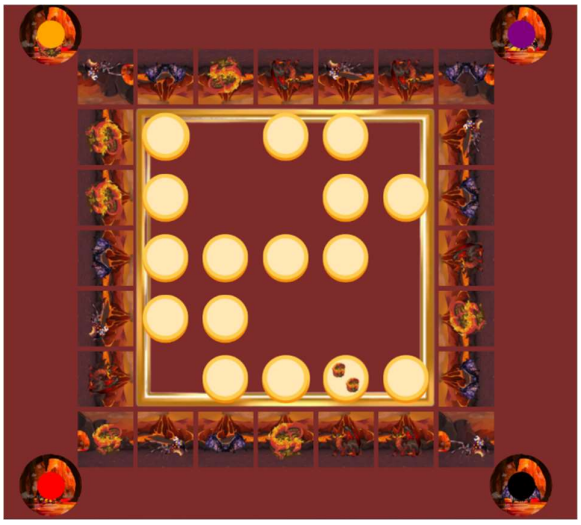


Figure 7: Flipped Dragon Card for Game State 2

As shown in the figures above, Figure 5 and 6 shows the board state for one game state while Figure 7 shows the board state for another game state. As shown between the two game states, the dragon card at the same position shows a different creature on it. Game state 1 shows a Pirate Dragon Card while Game state 2 shows a Salamander Dragon Card with 2 Salamanders on it.

Thus, the test to check if the dragon cards have randomized positioning is successful.

Executable Instructions

Libraries Needed

The third-party libraries needed for this game is JDK 22 and JavaFx version 22.0.1.

Instructions to Run the Game

1) Download and install the JDK 22 Development Kit 22:
   https://www.oracle.com/my/java/technologies/downloads/#jdk22-windows
   Download and install JavaFx version 22.0.1 SDK
   https://gluonhq.com/products/javafx/
2) Download the FieryDragonsGame.jar FILE
3) Run the jar file using Java(TM) Platform SE binary.

Instructions to build the executable

1. Create a new module and in the Public Static void main program call the main method from the main module of the javafx project.
2. Go to File -> Project Structure. Then go to the Artifact tab and press the "+" button.
3. Then press JAR -> from module with dependencies.
4. Then at the Main class choose the newly created main class. Then Press ok
5. At the Output Layout. Press the "+" button then choose File.
6. Find the extracted JavaFx folder then go into the Bin file and choose all the files that ends with .dll.
7. Click Apply then Ok.
8. Go to Build -> Build Artifact
9. Press Build and you will see the .jar file be created in a newly created directory call "out".