FIT3077: Software Engineering: Architecture and design

Semester 1, 2024

Sprint 3 Documentation

Group No: MA_Wednesday_04pm_Team690

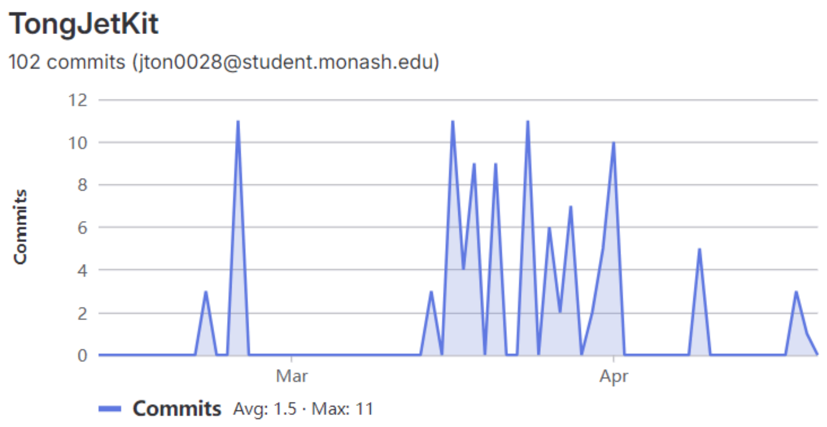
Prepared by:

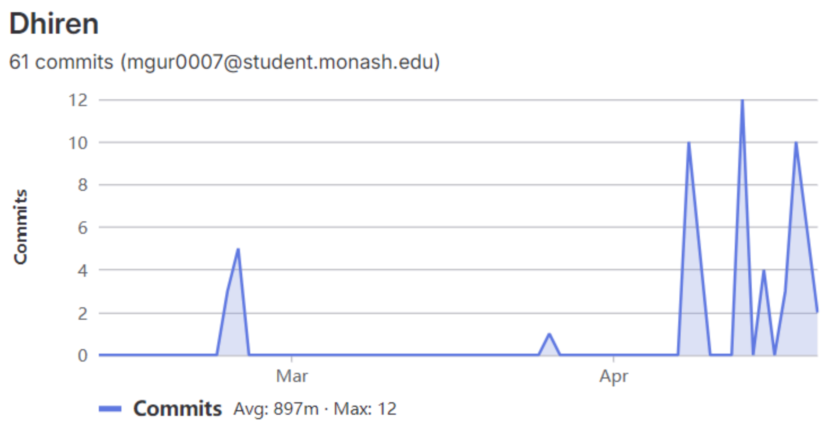
Tong Jet Kit	32632177	jton0028@student.monash.edu
Mandhiren Singh Gurdev Singh	32229828	mgur0007@student.monash.edu
Nisha Kannapper	31121993	nkan0018@student.monash.edu
Ong Chong How	33363102	cong0027@student.monash.edu

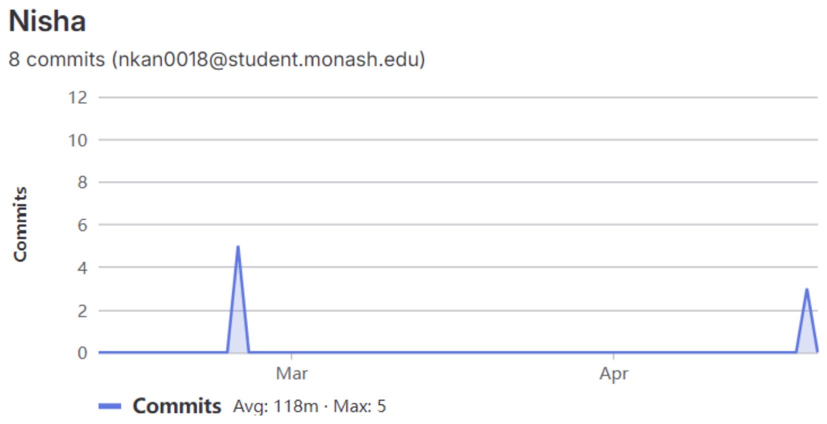
Content

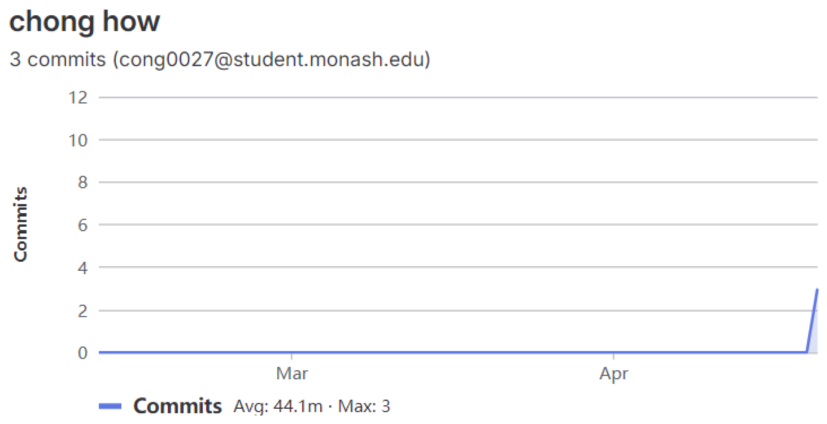
Contributor Analysis

Reminder: Ong Chong How is a sudden addition to our team, Due to time constraints and since his work in Sprint 2 was done Python, in this Sprint 3 he did not join in on the coding section but he helped out in the documentations which can be found in the figures below. However, he will be given coding tasks in Sprint 4.

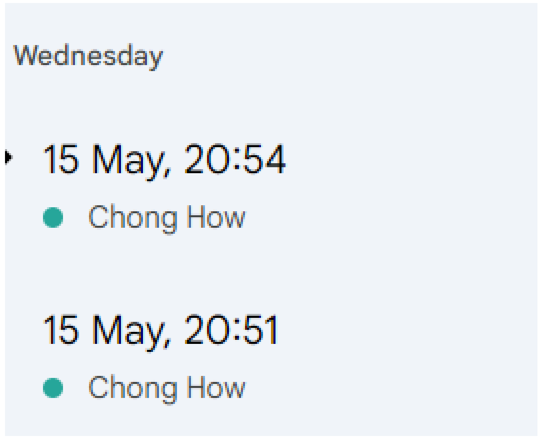








Ong Chong How Contribution on CRC Model documentation and Review Documentation


Wednesday

15 May, 20:54

● Chong How

15 May, 20:51

● Chong How

Quality Assessment Criteria

Below is a table that shows the quality assessment criteria that our team will be using for the review assessment. The table is separated into 5 columns:

1. Factor
2. Characteristics
3. Design Principles
4. Metric
5. Value

Factor	Characteristics	Design Principles	Metric	Value
Functional Suitability	Completeness	Test each scenarios for each function	Number of test cases pass	All test cases pass
	Correctness	All the basic classes	Number of classes	5 and above
Flexibility	Scalability	Scalability: No hard coded designs, ensure each function is dynamic and not static	Scalability: Number of hard coded code lines	1-10
Maintainability	Reusability Modularity	Utilisation of abstract classes	Number of abstract classes created	1-10
Interaction capability	User interaction	Has interactive UI components	Number of buttons/gesture detections	1-10

Review : Implementation (codes)

Mandhiren Singh

Key Game Functionality 1: Setting Up Board

Functional Suitability

The functionality of setting up the board created by Dhiren is not functional in correctness. This is due to the fact that the volcano card creation does not conform to the basic implementation of the game which has a fixed combination of the volcano card group tiles. Thus, the implementation of the gameboard is wrong already. Moreover the volcano card implementation will also allow 3 volcano card tiles to have the same creature on it.

Flexibility

The implementation is not flexible as there are a lot of hard coded lines. For example the valid positioning method is hardcoded and the volcano card and dragon cards are all hard coded too. For this sprint 2, it is ok that the dragon cards are hard coded as the basic game just has these few cards but the valid positioning should not be. Therefore, it is not flexible which will make it hard to extend different types of dragon card combinations in the future and to have a bigger board. However, this will be a technical debt for now.

Maintainability

The maintainability of the implementation is non-maintainable. This is due to the lack of abstract classes used. For example, the game cards are not extended from abstract classes which they should since they have similar attributes. This will then cause code duplication to occur which is not recommended as there might be inconsistencies in attributes and behaviours and will need to update all implementations.

Interaction Capability

There is interaction capability for this implementation as there is the start button where if you click it you will boot up the game screen and set up the game board.

Key Game Functionality 2: Flipping the Dragon Card

This game functionality is not implemented by Dhiren in Sprint 2.

Key Game Functionality 3: Moving the player based on the flipped Dragon Card

Functional Suitability

The functionality of this key game is complete and correct. The players can move based on the flipped dragon card and the number of creatures on the dragon card. The number of player token movement is correct as well. Hence, this key game functionality is implemented well and correct.

Flexibility

The implementation is flexible as the movement is based on the board size and coordinates in the gridpane. This makes it easy to implement more board sizes as the movement is not rigid and fixed to the basic game implementation of 24 volcano card tiles.

Maintainability

This game functionality implementation is not maintainable as the movement code is modularised into a method in the game rather than an interface class. This will then cause any movement action to reference the game board class to move it which sometimes is unnecessary.

Interaction Capability

There is interaction capability for this implementation. The user interface highlights the current player and the movement of the player token can be shown through a button click.

Key Game Functionality 4: Changing Turn to Next Player

Functional Suitability

The changing turn to next player functionality is complete and correct as we were able to see that the turn has changed from 1 player to another after flipping the wrong dragon card. The next player is highlighted with a golden ring and the previously active player will be unhighlighted when the next turn is triggered.

Flexibility

The implementation is flexible as the nextPlayerTurn() method works if more players are added to the players list in the game board class. Hence, it does not matter how many players are there as it will go through the player list to find the next player and switch to it by highlighting the next player and unhighlighting the previous player.

Maintainability

This game functionality implementation is maintainable as if needed to change turns to the next player, the next turn method can be just invoked to change the turn. Thus, it is modularable and reusable.

Interaction Capability

There is interaction capability for this implementation. The user interface shows the next turn changes through the highlight and unhighlighting of the current active player and next active player when the current active player flips a pirate dragon card or a different creature card. Also, if the position is occupied, the current player turn ends and the change of turn will occur.

Key Game Functionality 5: Winning The Game

This game functionality is not implemented by Dhiren in Sprint 2.

Tong Jet Kit

Key Game Functionality 1: Setting Up Board

Functional Suitability
The functionality of setting up the board is complete and correct. All the game cards are created and laid out correctly in the gameboard. The dragon card is scattered in the center and does not show any discrepancies between each dragon card. The cave cards are placed at the side with player tokens located on it.

Flexibility
The implementation is kind of flexible as there are a lot of hard coded lines for the cards. However, For this sprint 2, it is ok that the dragon cards are hard coded as the basic game just has these few card combinations. For the rest of the game, the volcano card creation is flexible as it uses the board size to determine the placement of the volcano card.

Maintainability
This implementation is maintainable as there is usage of abstract classes for the card objects which is the Card abstract class. This allows any future new card to be implemented easily as they can extend the abstract classes. This is because the card abstract classes implemented will allow all of its' child classes to inherit the common attributes and methods like image view and card symbol.

Interaction Capability
There is interaction capability for this implementation as there is the start button where if you click it you will boot up the game screen and set up the game board.

Key Game Functionality 2: Flipping the Dragon Card

Functional Suitability

The functionality of flipping the dragon card is complete and correct. All the game cards are created and laid out correctly in the gameboard. The dragon card is scattered in the center and does not show any discrepancies between each dragon card. The cave cards are placed at the side with player tokens located on it.

Flexibility

The implementation is kind of flexible as there are a lot of hard coded lines for the cards. However, For this sprint 2, it is ok that the dragon cards are hard coded as the basic game just has these few card combinations. For the rest of the game, the volcano card creation is flexible as it uses the board size to determine the placement of the volcano card. However for the flipping card action it is flexible as it uses a Flippable interface class for the flipping action.

Maintainability

This implementation is maintainable as there is usage of abstract classes for the card objects which is the Card abstract class. This allows any future new card to be implemented easily as they can extend the abstract classes. This is because the card abstract classes implemented will allow all of its' child classes to inherit the common attributes and methods like image view and card symbol.

Interaction Capability

There is interaction capability for this implementation as the dragon cards can be clicked by the player to flip it.

Key Game Functionality 3: Moving the player based on the flipped Dragon Card'

This game functionality is not implemented by Tong Jet Kit in Sprint 2.

Key Game Functionality 4: Changing Turn to Next Player

This game functionality is not implemented by Tong Jet Kit in Sprint 2.

Key Game Functionality 5: Winning The Game

This game functionality is not implemented by Tong Jet Kit in Sprint 2.

Nisha Kannapper

Key Game Functionality 1: Setting Up Board

Functional Suitability

The functionality of setting up the board is complete but not correct. Although all the game cards are created and laid out correctly in the gameboard. The generation of the volcano card is wrong as it does not conform to the basic implementation of the game which has a fixed combination of the volcano card group tiles. Moreover, the randomisation of the game cards does not meet the basic requirements as it does not show all the unique possible dragon cards. There are duplicate dragon cards which are wrong.

Flexibility

The implementation is not very flexible. This is because the randomization is not flexible but the creation of the card types are flexible. The randomization is done by listing out all the possible combinations rather than using the in build shuffle method. Hence, it is super rigid and unnecessary to be implemented this way.

Maintainability

This implementation is maintainable as there is usage of abstract classes for the card objects which is the Card abstract class. This allows any future new card to be implemented easily as they can extend the abstract classes. This is because the card abstract classes implemented will allow all of its' child classes to inherit the common attributes and methods like image view and card symbol. Moreover, there is also the Component abstract class which all the game objects extend from. This component class will represent objects that will be displayed in the UI which is flexible if there are extra things to show in the future like a new card or token.

Interaction Capability

There is interaction capability for this implementation as there is the start button where if you click it you will boot up the game screen and set up the game board.

Key Game Functionality 2: Flipping the Dragon Card

This game functionality is not implemented by Nisha in Sprint 2.

Key Game Functionality 3: Moving the player based on the flipped Dragon Card

Functional Suitability

The functionality of moving the player token based on the flipped dragon card is complete and correct. The player will move according to the flipped dragon card and will end their turn if they pick the wrong one or land on a tile with a player on it already. The player can move forward up to 3 spaces and move backward up to 2 spaces. Moreover, if reaching a full circle, the player will go back to their own cave.

Flexibility

The implementation is not flexible. This is because the movement is based on hard coded coordinates and indexing. This makes it hard to extend further if the game board size can be dynamically sized. Since it uses hard coded coordinates, the movement can only work on this 24 volcano card tile game board.

Maintainability

This implementation is maintainable as there is usage of abstract classes. For example, the Component abstract class is created which passes on all the behaviour of handling all UI aspects. This component abstract class is created to construct the graphic for each of the game objects to the UI. With this abstract class if in the future there is any new UI object we can then extend this class to set the UI for the object.

Interaction Capability

There is interaction capability for this implementation. There are demo movement buttons to simulate the player's movement during the game.

Key Game Functionality 4: Changing Turn to Next Player

This game functionality is not implemented by Nisha in Sprint 2.

Key Game Functionality 5: Winning The Game

Functional Suitability

The functionality of winning the game is not fully implemented. The program logic is correct, there is a check if the active player will win after moving, However, the methods to win the game are not invoked at all.

Flexibility

The implementation is flexible. The conditions to win the game are not hardcoded and ensures that the player has gone through a full circle and landed back to the cave before declaring the player is a winner.

Maintainability

This implementation is maintainable as the checking to win is implemented in the movement. If there are other ways to win the game then it has to code out its own winning condition code rather than calling a single method that will do it all. Hence, the checking for winning is not modularized.

Interaction Capability

There is no interaction capability for this functionality as the function is not fully implemented.

Chong How

Key Game Functionality 1: Setting Up Board

Functional Suitability

The functionality of setting up the board is not complete and not correct. Although all the game cards are created and laid out correctly in the gameboard UI, the volcano card, cave card and dragon card does not even show what creatures are on it. Moreover, the player tokens are not visible.

Flexibility

The functionality is not flexible as there are a lot of hard coded lines for the cards. However, For this sprint 2, it is ok that the dragon cards are hard coded as the basic game just has these few card combinations. However, the dragon card location is hard coded as well which makes it hard to extend if the board size increases.

Maintainability

The functionality is maintainable as there is usage of abstract classes and interfaces like drawable and movable. This then allows us to reuse attributes and behaviours or implement required behaviours only. For example, there is the DragonCard abstract class which allows future implementation of new dragon cards. However, it would be good if the VolcanoCard, CaveCard and DragonCard were extended from a Card abstract class. For the interfaces, the Drawable and Movable classes allow us to create more UI objects or movable objects.

Interaction Capability

There is interaction capability for this implementation as there are the user needed inputs before setting up the board.

Key Game Functionality 2: Flipping the Dragon Card

Functional Suitability

The functionality of flipping the dragon card is not correct and complete. A player cannot flip back the card and it uses the wrong method to determine if a player is flipping the dragon card. It should use the collidePoint function instead.

Flexibility

The functionality is flexible as there is usage of interface Flippable for the flippable dragon card. This then allows any future flippable dragon cards to be able to flip. Moreover, if there are other flippable objects then they can extend this interface to implement the flipping action.

Maintainability

The functionality is maintainable as there is usage of abstract classes and interfaces like drawable and movable. This then allows us to reuse attributes and behaviours or implement required behaviours only. For example, there is the DragonCard abstract class which allows future implementation of new dragon cards. However, it would be good if the VolcanoCard, CaveCard and DragonCard were extended from a Card abstract class. For the interfaces, the Drawable and Movable classes allow us to create more UI objects or movable objects.

Interaction Capability

The interaction capability is not complete as the user cannot flip back the dragon card. The user can flip but cannot unflip the dragon card. Moreover the text cursor for the user input is not shown which makes the user hard to notice if they can type. Moreover it does not allow the user to switch to fill in the age first before the name.

Key Game Functionality 3: Moving the player based on the flipped Dragon Card

This game functionality is not implemented by Chong How in Sprint 2.

Key Game Functionality 4: Changing Turn to Next Player

This game functionality is not implemented by Chong How in Sprint 2.

Key Game Functionality 5: Winning The Game

This game functionality is not implemented by Chong How in Sprint 2.

Review : Design (UML/Sequence)

Tong Jet Kit

UML Class Diagram

Functional Suitability

The UML class diagram is complete, as all of the key game functionalities are covered by the classes present in the diagram. The class diagram presents a structured representation of the game's components. Including FieryDragonsGameBordController, FieryDragonsGameBoard, Player and various types of cards.

In terms of correctness, it does seem to follow the game rules and logic. Each class includes methods relevant to its functionality, such as initialising the game board, flipping dragon cards and others. Since, all necessary gameplay methods are addressed with methods and classes, correctness can be determined.

Flexibility

It is very flexible, as the classes manage specific modularised functionalities which allows for scalability. For example, if we add new types or cards or Players, it should work without modifying the codebase too much as inheritance, interfaces and abstract classes are present in terms of the DragonCards.

The use of abstraction and inheritance allows for flexibility in extending the functionality of the game. For example, the Card class serves as a base class for different types of cards (CaveCard, DragonCard, VolcanoCard), enabling the addition of new card types with ease. This abstraction promotes code reuse and simplifies the integration of future card variants, enhancing the game's flexibility.

The modular design of the classes promotes flexibility by encapsulating related functionality within cohesive units. For instance, the Player class encapsulates player attributes and behaviours, allowing for easy modification and extension of player-related functionality without affecting other parts of the codebase. Similarly, the FieryDragonsGameBoard class encapsulates game board management, facilitating changes to board layout or rules without impacting other components.

The class diagram suggests dynamic behaviour, such as the ability to flip dragon cards and handle player movements based on game conditions. These circumstances or player actions. For example, the FieryDragonGameBoard class includes methods for adding cards to the board dynamically, allowing for variations in gameplay without requiring extensive code modifications.

Maintainability

This diagram displays maintainability since there is use of abstraction and inheritance. The use of abstraction and inheritance promotes maintainability by facilitating code reuse and simplifying modifications. For instance, the Card class serves as a base class for different types of cards, allowing for consistent handling of card-related functionality across the game.

The class diagram demonstrates a modular structure with clear encapsulation of functionality within classes. For example, the Player class encapsulates player-related attributes and behaviours, while the FieryDragonsGameBoard class encapsulates game board management. This modular design promotes maintainability by allowing developers to isolate and modify individual components without affecting the entire system.

Interaction Capability

There is interaction capability shown in this UML diagram. The FXMLController class likely serves as the controller for the game's user interface (UI), indicating a capability to handle user interactions. Methods such as onEndTurnButtonClicked() suggest the ability to respond to user actions, such as clicking on UI elements like buttons

The use of event listeners, such as onEndTurnButtonClicked(), indicates a capability to handle user-triggered events. These listeners can be attached to UI elements to respond to specific user actions, enabling interaction with the game's interface.

Methods like flipCard() in the DragonCard class suggest a capability for triggering game actions programmatically. These methods can be called in response to user interactions or other game events, facilitating dynamic gameplay behaviour.

The Player class represents individual players in the game, each with attributes and methods to facilitate interaction with game elements. Methods like highlight() and unHighlight(0 suggest the ability to visually represent player actions or states within the game interface.

Classes such as FieryDragonsGameBoard likely manage the overall state of the game, including player turns, card placements, and other game mechanics. Methods like nextTurn() indicate a capability to progress the game state based on player interactions or other game events.

Sequence Diagram: Board Setup

Functional Suitability

The sequence diagram captures all the steps in flipping the card which includes the animation for it. Moreover, it also shows how the dragon card is able to resolve the flipping through the flipcard method.

|

Flexibility

It is quite flexible, as the flip(gameBoard) method starts a flipping sequence then it is followed by a pause using PauseTrainsition. This means that animations and card logic are decoupled, which increases scalability, and allows the animations to be independent of the game logic.

However, the FlipCardPane is quite specialised which handles the flipping animation and the interaction of DragonCards. This specificity might show us a limitation in terms of reusing this class for other card types, perhaps.

Maintainability

The sequence diagram does show maintainability with the usage of modularized functions to create the game objects. For example the creation of each game card is separate and the method to create the UI for the game object is separate too. This makes it easier to use the method if needed to create another instance of the game objects again.

Interaction Capability

The sequence diagrams depict the sequential execution of methods such as initializeBoard(), addVolcanoToBoard(), addCavesToBoard(), initializePlayer(),addDragonCardsToBoard(), and setUpUI(). These method invocations indicate the capability to orchestrate the setup process and create the necessary game board elements in a predetermined order.

The sequence diagrams show the setup of Ui components, including the placement of volcano cards, cave cards, dragon cards, and player tokens on the game board. Additionally, the setDragonCardClickListener() method suggests the capability to handle user interactions, such as clicking on dragon cards to trigger the flip card action. This event handling capability enhances player engagement and interaction with the game interface.

Sequence Diagram: Flipping the Dragon Card

Functional Suitability

The functionality for setting up the board shown in the sequence diagram is complete as all the cards are created and randomised. Moreover, the UI of the card is also created by the game controller class which is correct since it is its' responsibility. It also groups up the 3 volcano cards into 1 group together which adheres to the game.

Flexibility

The sequence diagram indicates good flexibility because the methods are modularised and do not have multiple responsibilities, which increases scalability. For example, adding caves and volcano cards to the board have their own method instead of being lumped together. However, by looking deep into the code implementation, the flip card action is tied to an interface class. Thus, I believe it would be better if the class for the card is stated as Flippable instead of DragonCard to show the flexibility through the usage of interface class.

Maintainability

The sequence diagram does show maintainability as the flip card action is tied to a method call. Thus, if needed the dragon card needs to be flipped in other ways than clicking on the dragon card like through a keyboard key press event then the flipping card action can be shown by invoking the flipCard() method.

Interaction Capability

The sequence diagram depicts the triggering of the flip action when a player clicks on a dragon card represented by the FlipCardPane component. This event handling capability enables the game to respond to user interactions in real time, enhancing player engagement and immersion.

The sequence diagram involves the use of a PauseTransition object to control the timing of the flip animation. By specifying a duration of 2 seconds for the pause transition and setting an event handler to execute the flip action upon completion, the game controls the timing and execution of the flip animation, ensuring a smooth and consistent user experience.

Upon flipping the dragon card, the flipCard method is called, passing the active player and the game board as parameters. This method invocation updates the game state by flipping the specified dragon card and applying any associated effects or changes to the game board or player status.

Sequence Diagram: Moving the player based on the flipped Dragon Card

Functional Suitability

The sequence diagram depicts all the basic steps in handling player movement which includes the 3 scenarios if the dragon card is a pirate dragon card, if the dragon card is the correct dragon card or if the card is the wrong dragon card. However, the correctness is hard to test since it is not implemented as it is not his Key game functionality implementation in sprint 2. But by just looking at the sequence diagram it is correct.

Flexibility

It is quite flexible as the diagram shows that it is able to handle different scenarios based on the type of dragon card flipped. Therefore, it shows that the game logic is designed to be able to integrate different card effects, which suggest good scalability in terms of the game mechanics. Besides that, methods such as movePlayer(board, destination) is generic enough to handle any move, forward or backwards, which indicates flexibility to position the player.

Maintainability

The sequence diagram does show maintainability as the flip card action is tied to a method call. Thus, if needed the dragon card needs to be flipped in other ways than clicking on the dragon card like through a keyboard key press event then the flipping card action can be shown by invoking the flipCard() method.

Interaction Capability

The sequence diagram depicts the handling of the flip card action by the DragonCard object, invoking the flipCard method with parameters player and gameBoard. This event handling capability enables the game to respond to the flip card event triggered by player actions.

The flipCard method accesses game state information, such as the current location index and the board's location cards, to determine the next destination for the player token. By retrieving the card symbol from the next location card, the method calculates the final destination for the player token based on the dragon card's creature count and any special conditions, such as encountering a pirate dragon card.

The sequence diagram illustrates the movement logic for the player token, including determining the final destination based on the dragon card's creature count and move offset, and updating the player's current position on the game board. This movement logic ensures that the player token moves forward, backward, or remains stationary on the board based on the specific conditions specified by the flipped dragon card.

Sequence Diagram: Changing Turn to Next Player

Functional Suitability
The sequence diagram does show when a player ends their turn and how it is shown which is through the highlight method. Moreover, how the next player is obtained is observed through getting the player index from the array of players which simulate a circular array. Hence, it is complete and correct.

Flexibility
It is flexible because the sequence diagram allows us to change the active player and manage the player's state in a way that supports scalability. Methods like index(), get(), highlight(), unhighlight() are modularised which indicates that the design can account for changes like modifying how the turns are calculated or how the players can be highlighted without any major changes to the code.

Maintainability
The sequence diagram does show maintainability as the flip card action is tied to a method call. Thus, if needed the dragon card needs to be flipped in other ways than clicking on the dragon card like through a keyboard key press event then the flipping card action can be shown by invoking the flipCard() method.

Interaction Capability
The sequence diagram depicts the handling of the end turn action by the FieryDragonGameBoardController, invoking the endTurn method. This event handling capability enables the game to respond to the end turn event triggered by the player's action of choosing the dragon card with the wrong creature.

The endTurn method accesses and manipulates the game state to transition to the next player's turn. This involves determining the index of the active player, retrieving the next player in the player list, unhighlighting the current active player, highlighting the next player, and setting the next player as the active player. These actions ensure proper management of the game state during the turn transition process.

The sequence diagram illustrates the interaction capability for highlighting and unhighlighting players to visually indicate their turn status. The highlight and unhighlight methods are called to apply visual cues to the active and inactive players, respectively, facilitating player identification and enhancing the user interface's clarity.

Sequence Diagram: Winning The Game

Functional Suitability

The sequence diagram displays all the scenarios such as if the active player is in its cave card or if its not and also how to resolve it by calling the appropriate methods like displayWinningMessage(). However, it is hard to determine the correctness as its not implemented but by the looks of the sequence diagram flow it is correct.

Flexibility

It is flexible, as the diagram allows for handling game logic variations. This is because the diagram shows a conditional flow where the game checks if the active player has reached their home cave in reachHomeCave(), by using checkPlayerLocation(). So, if we were to add new rules like needing a specific card to win or achieving other objectives, they can be integrated into the conditional flow without redoing the entire winning process.

Besides that, methods are mostly modular which enhances scalability. For example, checkForWin() starts the win check process and starts the sequence to check for the win only. This method is high-level, and we could modify it to include more preliminary checks or adding more win conditions.

Maintainability

The sequence diagram depicts the sequence of method calls to determine if a player has won. Each method call represents a separate functionality, and the checkPlayerLocation method can be used in multiple scenarios, making it modular and reusable.

Interaction Capability

The sequence diagram depicts the handling of the check for win action by the FieryDragonGameBoardController, invoking the checkForWin method. This event handling capability enables the game to respond to the player's action of reaching their home cave and potentially winning the game.

The checkForWin method accesses the game state, specifically the player's current location on the game board, to determine if the win condition has been met. This involves checking whether the active player has reached their home cave after traversing the entire board, indicating a successful completion of the game.

If the win condition is met (i.e., the player reaches their home cave), the displayWinningMessage method is invoked to notify the players of the game's outcome. This interaction capability provides feedback to the players, informing them of the game's conclusion and the winning player's achievement.

Ong Chong How

UML Class Diagram

Functional Suitability

The sequence diagram comprehensively delineates the requisite classes and methods essential for configuring the board and facilitating key game functionalities. Nevertheless, it also reveals the presence of superfluous classes such as GameSetup and PlayerSetup, whose functionalities can be amalgamated into a singular entity. By consolidating these entities, the design could achieve a more streamlined and efficient structure, thereby enhancing overall system coherence and minimizing unnecessary complexity.

Flexibility

The design exhibits commendable flexibility through the strategic utilization of interfaces such as Drawable, Flippable, and Movable. This deliberate design choice facilitates the decoupling of specific actions from concrete class implementations, thereby enabling seamless integration of additional game elements without necessitating modifications to the existing codebase. By mandating the implementation of these interfaces, any class requiring such behavior can readily conform, fostering extensibility and promoting maintainability.

Moreover, the inclusion of abstract classes like DragonCard exemplifies a scalable framework for diverse card functionalities. By defining a foundational structure through the DragonCard abstraction, the design facilitates the creation of new DragonCards via subclass extension. This approach ensures scalability by accommodating the integration of novel card functionalities without introducing disruptions to the established codebase.

Maintainability

This diagram shows maintainability since there is use of abstraction and inheritance. The abstraction and inheritance used facilitates code reuse and simplifies modifications. For instance, the DragonCard abstract class is the basis for the two types of dragon cards, which ensures reusable functionality related to the handling of the dragon cards.

The class diagram has a modular structure with all functionalities encapsulated within interfaces and classes. For example, the Drawable and Flippable interfaces contain methods used for displaying elements or flipping elements, which allows for a range of classes to reuse functionality.

Interaction Capability

The GameSetup class is responsible for managing the setup of the game, including generating players, drawing player details, updating player information, setting up the game board, and running the game. The methods provided in the class enable interaction with the game setup process, such as initialising players, configuring game parameters, and initiating gameplay.

The PlayerSetup class handles player-related setup tasks, such as displaying player details, prompting for the number of players, and collecting player information. The methods in this class facilitate interaction with players during the setup phase, allowing for the customization of player attributes and preferences.

The GameBoard class represents the game board, including tiles, caves, volcano cards, and player positions. It provides methods for generating the game board, handling mouse clicks, creating animals, generating dragon cards, drawing the board, and determining tile types. These methods enable interaction with the game board, such as mouse input for player actions and updating the board state during gameplay.

Sequence Diagram: Board Setup

Functional Suitability

The sequence diagram meticulously captures the comprehensive setup process for initializing the game board. It systematically instantiates all requisite game objects, including players, dragon cards, cave cards, and volcano cards, followed by the rendering of the corresponding user interface (UI) elements. Notably, it leverages the Tile class to effectively aggregate the volcano card tiles, thereby enhancing organizational coherence within the game board structure. This methodical depiction underscores the meticulous attention to detail inherent in the board setup phase, ensuring a robust foundation for seamless gameplay execution.

Flexibility

The sequence diagram highlights a moderate-to-low of flexibility due to its constraint on the number of players, set between 2 and 4 inclusively. This limitation suggests that accommodating additional players or implementing a single-player mode would necessitate modifications to the sequence logic within the alternative block, indicating a degree of rigidity in adapting to varying player counts.

Conversely, concerning scalability, the game components such as tiles, volcano cards, and dragon cards are organised within lists, demonstrating an adaptable architecture. This design choice facilitates straightforward scaling by adhering to consistent patterns for object creation and storage. Consequently, the system can readily integrate new components, such as additional card types, without extensive alterations, thus enhancing its capacity for expansion and diversification.

Maintainability

The diagram clearly separates the responsibilities between different components such as PlayerSetup, GameSetup, and GameBoard. This separation ensures that each component handles a specific aspect of the game initialization, making it easier to locate and modify code related to player setup, game board creation, or game start procedures.

The diagram shows the reuse of methods such as generate_gameboard(), create_tiles(), create_volcano_cards(), create_caves(), and create_dragon_cards(). Reusing methods promotes DRY (Don't Repeat Yourself) principles, reducing code duplication and making maintenance easier. If changes are needed in the creation process, they can be made in one place.

Interaction Capability

The sequence diagram starts with the player attempting to start the game, followed by the game engine prompting the player to input the number of players. This interaction capability enables the game to receive user input and respond accordingly, facilitating player engagement and customization of game parameters.

The game engine checks the validity of the number of players entered by the player. If the input is less than 2 or more than 4, the game initialization process halts, indicating a requirement violation.

This validation ensures that the game adheres to predefined constraints and requirements, enhancing the reliability and integrity of the gameplay experience.

Sequence Diagram: Flipping the Dragon Card

Functional Suitability

The sequence diagram is deficient in depicting the subsequent method call following the flipping of the card, solely delineating the animation aspect. This omission renders the sequence incomplete and inaccurate due to the absence of the resolving action. Nonetheless, it accurately and comprehensively captures the UI component associated with flipping the card. Therefore, while the graphical representation of the card-flipping process is correct and exhaustive, the overall sequence diagram lacks completeness in portraying the subsequent method invocation post-flip.

Flexibility

The sequence diagram exhibits a commendable level of flexibility, notably in its handling of card flipping through the evaluation of the "faced_up" attribute within the DragonCard. This design choice underscores the adaptability of the game's UI logic, as it can readily accommodate various interactions with game elements in a scalable manner. The existing codebase can seamlessly integrate additional functionalities, such as imposing rules upon card flipping, by extending the logic governed by the "faced_up" attribute check.

Furthermore, the diagram delineates a coherent pathway of interactions, tracing from the player's click on the card, through the flipping process, and culminating in the rendering of updates. This clear interaction flow implies that augmenting the system with new types of interactions, such as cards featuring additional actions upon flipping, can be effectively managed. This could be achieved by either appending new sequences to handle these interactions or by modifying the modularized flip method to incorporate supplementary effects, thereby fostering extensibility and ease of maintenance within the codebase.

Maintainability

The modularity and separation of concerns are evident in the distinct responsibilities assigned to different components. The Player initiates the action, GameSetup initializes the game components, GameBoard handles the core logic of flipping the card, and DragonCard manages the card's state. This encapsulation ensures that modifications or extensions to the card flipping logic can be made without affecting other parts of the system.

The design is scalable and extensible, allowing new features to be integrated with minimal changes. Event handling, indicated by handle mouse click, supports the addition of new event types or actions. The diagram also includes a check for whether the card is already flipped, preventing redundant actions and potential errors. State management ensures the game state remains consistent, reducing the likelihood of bugs.

Interaction Capability

The sequence commences with players exercising agency by selecting any dragon card presented on the game board, thereby engaging with the game interface and exerting influence over gameplay dynamics. Subsequently, the game engine initiates an assessment to ascertain the current state of the targeted dragon card. If the "faced_up" attribute of the selected dragon card evaluates to false, signifying that the card remains unflipped, the game engine proceeds to execute the card-flipping operation. This pivotal functionality empowers players to uncover the concealed content of the dragon card, thereby enhancing their strategic decision-making and immersive engagement with the gameplay experience.

Sequence Diagram: Moving the player based on the flipped Dragon Card

Functional Suitability
The sequence diagram is incomplete as it solely delineates the scenario wherein players advance by one space. The absence of further movement scenarios, such as diagonal movement, backward movement, or movement spanning multiple spaces, renders the diagram insufficient in capturing the full spectrum of possible player actions on the game board. Consequently, to provide a comprehensive depiction of gameplay dynamics, additional sequences detailing various movement scenarios should be incorporated into the diagram.

Flexibility
The sequence diagram exhibits dynamic behavior by eschewing hardcoded values in favor of utilizing attributes associated with cards and the game board for validation and decision-making processes. This design approach fosters adaptability and scalability, as the system's behavior is contingent upon the inherent properties of game elements rather than rigidly predefined constants. By leveraging card and board attributes during checks, the system accommodates variability in game configurations and facilitates seamless integration of new elements without necessitating extensive modifications to the underlying logic. This dynamic approach enhances the system's robustness and flexibility, enabling it to effectively respond to evolving gameplay scenarios.

Maintainability
The sequence diagram leverages the Movable interface, thereby enabling diverse game components to inherit the capability for movement and be recognized as instances of Movable. This strategic design choice enhances the system's extensibility and adaptability by establishing a standardised protocol for handling movement across various game elements. By adhering to the Movable interface, potential additions to the game, such as new components or entities, can seamlessly integrate with existing movement mechanisms without necessitating extensive modifications. This design promotes code reusability, simplifies maintenance efforts, and facilitates the incorporation of novel gameplay elements, thereby fostering a more dynamic and scalable game environment.

Interaction Capability
The sequence diagram is triggered by the flip of compatible Flippable cards, initiating an assessment of interaction capabilities. This mechanism allows for the evaluation of potential interactions between game elements following card flips, enabling dynamic gameplay experiences. By integrating this functionality, the system can effectively determine and execute relevant interactions based on the attributes and states of Flippable cards, enhancing the depth and immersion of gameplay scenarios.

Sequence Diagram: Changing Turn to Next Player

Functional Suitability

The sequence diagram pertaining to this game functionality is both comprehensive and accurate, as it effectively illustrates the process by which the game transitions to the next player's turn. Additionally, the diagram appropriately specifies that all dragon cards will be reverted to their original, face-down state prior to the commencement of the subsequent player's turn. This meticulous depiction ensures the seamless progression of gameplay while maintaining consistency and fairness in the game mechanics.

Flexibility

The sequence diagram demonstrates a commendable degree of flexibility in managing player turns. By iteratively cycling through players and updating the current player, the design showcases adaptability to accommodate an expanded pool of players, thereby enhancing scalability.

Furthermore, the inclusion of a reset mechanism for dragon cards—ensuring their return to an initial, face-down state at the conclusion of each player's turn—underscores the versatility of the design. This feature allows for the accommodation of a reset state, which is integral to gameplay dynamics, and can be adjusted as necessary to align with evolving gameplay requirements or variations. Overall, these aspects of the design contribute to its flexibility and suitability for potential modifications or expansions in the future.

Maintainability

Responsibilities are encapsulated within specific components, promoting encapsulation and modularity. The Player and GameSetup handle player-specific logic, such as determining the next player. The GameBoard manages the overall game state, including updating the current player and resetting dragon cards. The DragonCard manages the state of the cards, ensuring they are ready for the next player's interaction. This modular approach allows changes to be made to one part of the system without affecting others, enhancing maintainability.

The loop structure ([For dragon_card in range(len([dragon_cards]))]) indicates that the system can handle multiple players and cards efficiently, ensuring scalability. The use of lists to manage players allows for easy extension, accommodating varying numbers of players without significant changes to the underlying logic. The methods depicted are reusable, such as the clear method for updating the current player and the centralized process for resetting cards, ensuring consistency and reducing code duplication.

To further enhance maintainability, it is recommended to implement comprehensive documentation explaining the purpose and function of each method, robust error handling to manage unexpected scenarios, and thorough unit testing to catch potential issues early. Descriptive naming conventions and textual explanations within the sequence diagram improve readability, making it easier for developers to follow the logic. Regular code reviews and refactoring will help maintain simplicity and clarity. By following these guidelines, the sequence diagram and the associated code will remain maintainable, adaptable, and robust, ensuring a smooth and enjoyable gameplay experience for users.

Interaction Capability

The sequence commences with the initiation of the current player's turn, signaling their opportunity to enact a move on the game board. This functionality underscores the sequential progression of player turns, thereby upholding the principles of fairness and equilibrium throughout gameplay.

Subsequently, the game board undertakes an examination to ascertain whether the animal depicted on the flipped dragon card aligns with the animal attributed to the current player's position on the game board. This comparison dictates the player's eligibility to execute a move, contingent upon the fulfilment of the condition stipulating a match between the dragon card and the player's assigned animal.

The sequence then proceeds to evaluate the occupancy status of the destination tile. Depending on the outcome of this assessment, the player's turn may culminate either upon successfully relocating to the new tile or encountering an impediment due to the tile's existing occupancy. This interactive mechanism effectively governs the flow of gameplay turns, ensuring adherence to established rules governing movement and tile occupation.

Sequence Diagram: Winning The Game

Functional Suitability

The sequence diagram is indeed comprehensive and accurate, as it effectively delineates both winning scenarios and incorporates checks to ascertain if a player's movement results in reaching their home cave. By encompassing these critical gameplay elements, the diagram provides a holistic representation of the conditions required for victory and the mechanisms governing player movement. This completeness ensures a thorough understanding of the game's logic and facilitates effective implementation and testing of the corresponding functionalities.

Flexibility

The sequence diagram demonstrates flexibility by employing dynamic checks to determine if a player can move to a CaveCard. This dynamic assessment relies on attributes associated with both the Player and the CaveCard, rather than fixed criteria, allowing for adaptable gameplay experiences. By basing the movement eligibility on variable attributes, such as player capabilities and cave card conditions, the system accommodates diverse gameplay scenarios without requiring rigid, predefined rules. This dynamic approach enhances the game's versatility and scalability, enabling it to adjust to different player configurations and evolving game states.

Maintainability

The sequence diagram demonstrates maintainability by utilizing the abstract class AnimalType to ascertain if a player's creature type matches that of a cave. This design choice promotes code maintainability by centralizing the comparison logic within a single abstract class, facilitating easier modification and extension in the future. By encapsulating the comparison logic in AnimalType, the system adheres to the principles of abstraction and encapsulation, enhancing readability and ease of maintenance. This approach minimizes code duplication and reduces the likelihood of errors when implementing and updating creature type comparisons throughout the game.

Interaction Capability

The absence of UI components and the reliance on player movement to trigger game end underscore a streamlined approach to interaction capability assessment within the sequence diagram. By aligning game termination with player movement rather than relying on UI events, the design enhances simplicity and cohesion. This streamlined process reduces complexity and potential points of failure, promoting clarity and efficiency in interaction assessment. Additionally, coupling game end with player movement provides a clear and intuitive mechanism for determining when gameplay objectives are achieved, contributing to a seamless and intuitive user experience.

Mandhiren Singh

UML Class Diagram

Functional Suitability

There is a lack of completeness of functions as not all required class methods are present in the classes to make the key game functionalities work. The movement and change of player turn is complete and correct, but others like card flipping and winning the game have not yet been implemented in full. This lack of completeness compromises the game's functionality as not all class methods required to support key game features are present.

Flexibility

The UML Diagram is moderately flexible, as the DragonCard can be extended to include more "special" DragonCards in the future due to the inheritance nature of it. However, the game board class is not that flexible because the boardSize is made as a fixed attribute (fixed value) but not included in the constructor so we cannot increase the boardSize easily.

Maintainability

The design is not really maintainable, because there are no abstract classes or interfaces that can be used to reduce repeated code. For the movement, it is handled directly in the FieryDragonsGameBoard class and there does not exist a "Movable" class which complicates any modifications related to the movement mechanics. The lack of an abstract class for card objects forces developers to repeatedly write out card attributes and methods for each new card, significantly increasing the potential for errors and redundancy.

Interaction Capability

The system handles interaction capabilities adequately, as there are controllers which manage the UI changes as a response to player actions. Functions like card flipping are managed by the GameController class, which updates the UI dynamically based on user interactions. This setup ensures that the game responds appropriately to user inputs, such as moving tokens or flipping cards, which enhances the player's engagement and the interactive quality of the game.

Sequence Diagram: Board Setup

Functional Suitability

The board setup is incomplete, because only the DragonCard randomization is included in the sequence diagram, although in the code everything is fully implemented. The VolcanoCard randomisation is not present in the sequence diagram.

Flexibility

It is not really flexible, as the sequence diagram uses createDragonCards() to instantiate the cards one by one, in a rather hard coded-fashion. So, if more cards are to be implemented in the future, we'd have to hardcode them in for the randomization and reference to the card ImageViews. The creation of DragonCards is done in a single method, which means that if further cards are created, there is a need to modify the codebase.

Maintainability

It is moderately maintainable, because the hardcoded areas are contained within modularised blocks of codes (methods) in which the logic can be easily modified within those methods themselves. For example, we can change it from being hardcoded to non-hardcoded within the createDragonCards() method itself.

Interaction Capability

Interactive UI components are present, as the methods like createDragonCards() and placeDragonCardsRandomly() directly update the UI when the card flip button is clicked.

Sequence Diagram: Flipping the Dragon Card

Functional Suitability

It is complete and correct, because all methods required for flipping the card and UI setup have been mentioned in the sequence diagram to make the card flipping work. It seems to be able to comprehensively capture the actions required for flipping the cards and setting up the UI. By including all necessary methods for these tasks, the diagram suggests that the essential functionalities for the flipping of the card are all accounted for.

Flexibility

It is relatively flexible, as the getCardDetails() method is supposed to work on the extended DragonCard classes. So, more DragonCard child classes can be implemented all while having different game logic "details". Also, flexibility is present, considering the method allows for any DragonCard instances to be flipped, allowing for future subclasses to also have the same flipping functionality

Maintainability

It is relatively maintainable as there are no hard-coded bits in the method calls. However, the code has not been implemented in full yet so we need to ensure that we do not hard-code the methods. Besides that, the flipCard method being a DragonCard method restrains it from being used by other potentially flippable cards in future iterations of the game.

Interaction Capability

The UI interaction capabilities are present, as the movePlayer() method directly moves the Player on the board after the card is flipped and the visual changes are realised on the user interface (UI). However, there is no mention of an event that can trigger the flipping capabilities is present in the diagram. Without a clear representation of the clicking event, developers and stakeholders may question how the interaction between the player and the card is initiated and executed within the game.

Sequence Diagram: Moving the player based on the flipped Dragon Card

Functional Suitability

The sequence diagram is moderately complete and correct, as the method calls for movement have all been directly tested and implemented in the codebase. This includes niche game functionalities like ending the player's turn if the position is occupied or a pirate dragon card is flipped. However, it does not mention the movement if the dragon card is a pirate dragon card.

Flexibility

The sequence diagram shows relatively decent flexibility. The movement is dynamically coded as it uses the board size to calculate the cells for the player to move on. Therefore, if we increase the board size, the maths to calculate the player movement should work. The movement is also flexible as it can accommodate the different board sizes by using the getCurrentRow() and getCurrentColumn() method to get the location of the destination.

Maintainability

It is moderately maintainable as there is reusability since the dragon card is an abstract class so other dragon cards can also have access to the movement card after acting on it. However, the maintainability is slightly compromised as there are no abstract classes or interfaces used, such as perhaps a "movable" interface that lets different types of objects be movable for future implementations.

Interaction Capability

Interaction with the UI is mostly done through the movePlayerForward(), placePlayer() and nextPlayerTurn() methods, where they update the UI based on their corresponding functionalities. There is no direct interaction capability with the user explicitly as it is done when the player flips a DragonCard so this is the best it can do.

Sequence Diagram: Changing Turn to Next Player

Functional Suitability

The sequence diagram is fully complete and correct, as the method calls have all been directly tested and implemented in the codebase. The change of turn to the next player is done with the nextPlayerTurn() method which simply updates the current player. It also goes around when player 4's turn is over, to allow player 1 to continue his turn, in a circular-queue fashion. The collisions are also handled as there is a boolean check "isPositionOccupied()" that checks if a tile is occupied by another player. If it is, the player's turn ends. Also, if the PirateDragonCard is flipped, the turn ends as well.

Flexibility

The code is flexible, as the update in currentPlayer is calculated directly based on the list size in the nextPlayerTurn() method call. This means that we are able to always update the currentPlayer regardless of the number of players should it be increased in the future. It is also dynamically coded, as the number of players used to determine the next player is not a hardcoded number, but based on the number of players present in the game

Maintainability

There is a moderate level of maintainability, as the method used to end a player's turn (nextPlayerTurn()) can be reused in different sections or can be set to trigger with other actions, making it modular. This modular approach does promote maintainability by encapsulating the turn-ending functionality within a single method.

Interaction Capability

The interaction capability is also present because it uses a "Different Creature Card Flip" button to conclude player turn, which has a listener called onEndTurnButtonClicked() that updates the currentPlayer and changes the player's turn when it is clicked.

Sequence Diagram: Winning The Game

Functional Suitability

The sequence diagram is complete, as the methods required are fully included in the diagram. However, the correctness may change as the methods are not fully implemented yet. The functionalities in question are to check after each flip, determining the winning factor and displaying a win message if win or go to the next turn if not.

Flexibility

The diagram is considered flexible, particularly in how the checkForWin() method is described. This method's flexibility stems from its ability to dynamically calculate the winning factor based on the board size, rather than relying on a static rule set. This design choice allows for adjustments in how victories are determined as the game evolves or if new rules are introduced.

Maintainability

Maintainability seems to be a slight concern, with the absence of abstract classes or interfaces which could complicate future efforts to update or refactor the win-checking logic. However, Jet Kit suggests that the method's modular design inherently supports maintainability, providing a solid foundation for future enhancements or modifications to the win determination logic.

Interaction Capability

In terms of interaction capability, the sequence diagram includes the displayWinMessage() method, which directly interacts with the UI to notify players of the game outcome. This ensures that players receive immediate feedback when a win condition is met, enhancing the user experience by making game outcomes clear and engaging. However, while the diagram shows the process following a card flip, it does not involve direct interaction with the user beyond this automated response.

Nisha Kannapper

UML Class Diagram

Functional Suitability

It seems to be mostly correct and complete (however, non-key game functionalities are not implemented yet in the codebase), the cardinalities are correct and complete as well. However, in terms of the associations, I think more compositions and aggregations are required so as to increase the specificity of the UML Class Diagram. Therefore, it is moderately suitable.

Flexibility

It is relatively flexible as there is a clear separation between the controllers, UI and logic following the Model-View-Controller (MVC) method. The UI is handled by the controllers and the logic is handled by the Board class. This separation allows us to modularise the code into specific compartments, where each developer can fully focus on developing their specific part in the future, which allows the game to be scalable to a larger team and codebase.

Maintainability

It is relatively maintainable as the classes are generally dynamically coded except for the randomization of DragonCards though the Board class is hard-coded. There is also a presence of abstract classes like the Card and Component class. So, if the game were to have new dragon cards or UI components in the future, then the developers will be able to simply extend these classes so as to integrate it into the game. For example, if a Special Dragon Card is to be added in the future, we can just extend the Card class and inherit all common attributes there but then implement functionality specific to that Special Dragon Card.

Interaction Capabilities

The interaction capabilities are present, as per the Component class in terms of setting up UI components like the shape and images for the DragonCards. Since there are a lot of UI components like buttons and shapes, we can safely say that the users are able to interact well with this system as per the UML Class Diagram.

Sequence Diagram: Board Setup

Functional Suitability
The board setup is complete, as the DragonCards, Players and Board are all setup properly according to the sequence diagram, where method calls for all of these are present. In terms of correctness, all setup methods required have been included in the sequence diagram.

Flexibility
It is relatively flexible as the classes are generally dynamically coded. However, the randomization of DragonCards through the Board class is hard-coded. Therefore, if we were to scale the application, we would have to individually code the locations of the DragonCards for randomising it, when it should have been dynamically coded. There is, however, some flexibility in scaling the application with the use of inheritance in terms of the VolcanoCard, CaveCard and CreatureCard. This modularisation allows for developers to scale the app if new Cards are to be added in the future in terms of their functionalities.

Maintainability
It is moderately maintainable, with the hardcoded randomisation of DragonCards bringing the maintainability down a notch. However, purely looking at the sequence diagram, we can see that it is somewhat maintainable as the cards there inherit from a parent class where the common methods and attributes are shared amongst cards, but specific functionalities are implemented in the child card classes. So, it will be maintainable in this aspect as the developers would just be able to extend this class. Besides that, it is relatively maintainable as the initialisation of board components are modularised in the methods, and we can easily change the hardcoded components in their respective method blocks.

Interaction Capabilities
The interaction capabilities are present, as the initialisation oF VolcanoCards, DragonCards, CaveCards and Players all work in tandem to update the UI based on the interaction by the Player which is to “start the game”

Sequence Diagram: Flipping the Dragon Card

Functional Suitability
The sequence diagram is not present and has not been completed in Sprint 2, therefore it is not functionally suitable.

Flexibility
The sequence diagram is not present and has not been completed in Sprint 2, therefore it is not flexible.

Maintainability
The sequence diagram is not present and has not been completed in Sprint 2, therefore it is not maintainable.

Interaction Capabilities
The sequence diagram is not present and has not been completed in Sprint 2, therefore it is not capable of interactions.

Sequence Diagram: Moving the player based on the flipped Dragon Card

Functional Suitability

The methods for player movement seem to be complete as the logic based on the sequence diagram initially checks for the Player win, then the processMove() method allows the Player's coordinates to be updated on the GridPane.

However, correctness is not guaranteed because the getHasWon() method is called before the processMove() method which may cause it to pre-emptively check for win when the Player has not yet processed his move.

Flexibility

It is flexible, as the getHasWon() method should work every single time a Player moves, even if more Players are instantiated and are playing the game.

Besides that, in terms of the processMove() method, it is also decently flexible as it can be called to update the coordinates of the Player base based on the getMoveCounter() method to determine the next location of the player. The getMoveCounter() method is also flexible because it should determine the movement (either forward or backward) based on any DragonCard child classes. This also makes the game scalable.

Maintainability

It is reusable as the method to movePlayer can be called in any other situation that needs a player movement. Thus it has reusability and modularity due to method abstraction. It is also decently maintainable because there seems to be a decoupling of classes like the Player class and the CaveCard, so future extensions to these classes will be easier due to the modularity and developers can focus on the parts of the code where it is modularised.

Interaction Capabilities

The interaction capability is seen through the player's movement across the volcano card when a dragon card is flipped. Besides that, the buttons on the screen that is connected to the methods like movePlayer() make it so that the user can interact well with the application.

Sequence Diagram: Changing Turn to Next Player

Functional Suitability

The sequence diagram is not present and has not been completed in Sprint 2, therefore it is not functionally suitable.

Flexibility

The sequence diagram is not present and has not been completed in Sprint 2, therefore it is not flexible.

Maintainability

The sequence diagram is not present and has not been completed in Sprint 2, therefore it is not maintainable.

Interaction Capabilities

The sequence diagram is not present and has not been completed in Sprint 2, therefore it is not capable of interactions.

Sequence Diagram: Winning The Game

Functional Suitability

The sequence diagram does not clearly explain how the winning game works as there are missing condition explanations and lack of textual explanations. Thus, it is hard to determine the completeness and correctness of the board setup. For example, the alternative blocks in the sequence diagram does not explain how the conditions work.

Flexibility

The sequence diagram shows limited flexibility as the game logic is heavily dependent on the specific sequence of method calls and conditions in terms of the "alternative" blocks of code. This could hinder the ability to introduce new features. For example, additional player actions or alternative game rules that require intermediary method calls.

However, the part where there is a player collision check ("if moving player's playerNum == this player's playerNum"), it is rather scalable because it should work even if we increase the number of players in the game. It just loops through all the Players and does the exact same thing.

Maintainability

It is moderately maintainable as modularity in terms of methods like processMove() and getCoords() suggest that parts of the game logic are encapsulated.

However, the maintainability might be compromised by the complex interdependencies illustrated, like the conditionals that manage game states and player interactions. Methods such as canWin() and setHasWon() might imply that the logic is kind of intertwined, which needs careful management to ensure consistency in the game. There are also no abstract classes or interfaces in this sequence diagram.

Interaction Capabilities

The interaction capability is seen through the player's movement across the volcano card when a dragon card is flipped, as per with the movement sequence diagram. However, there is no update of UI components showing the details of the move, therefore it is moderately interaction-capable.

1. Outline how the creation of the tech-based prototype for Sprint 3 (that is to cover the entire game functionality) is to be performed by answering the questions:

Jet Kit: UI creation method and Flipping

Rys: Next turn and Win condition

Dhiren: UI object, movement code

Chong How: Due to chong how being a late addition to our group, and his codebase being in a different programming language from ours. We are choosing not to try to convert anything to java and will just be assigning a different task for him. He will be handling Documentation

Review : Design Rationale

Mandhiren Singh

Functional Suitability

The functional suitability can be seen through the relationship between the FieryDragonGameBoard and Player class and the GameScreenController and FieryDragonsGameBoard. It is correct that the FieryDragonGameBoard should have an aggregate relationship to the Player class. This is because the game board should always know its' player, especially the current active player while the existence of players is not strictly dependent on the game board itself. This will decouple the life cycles between the 2 objects which will open up a ton of new features to implement. Moreover, it is correct as if the game board has no idea what player it has then it would be hard for the game functionality like player movement to work as intended.

Besides that the relationship between the FieryDragonGameBoard and the GameScreenController is correct and suitable. The GameScreenController is a class that will function specifically to control the game's UI and therefore has a FieryDragonsGameBoard attribute. Thus, it needs to have a relationship with the gameboard which contains all the game's data. Without it, then there is no game. The design rationale also states that the relationship is a composition relationship which I believe is not true. This is because the gameboard can exist without the controller class as it handles the behaviour of the data of the application and it does not need to know who will use itself. Moreover, the gameboard is considered a Model if we view it based on the Model View Controller architecture. A Model should be built independent from the View and the Container. So, the type of relationship is wrong since composition states that both classes depend on each other but the gameboard should not depend on the controller class. However, it is true that there is a relationship between them as the controller needs to know the gameboard so it can inform the gameboard of any request to respond to.

Flexibility

For the design rationale, it mentions the aggregation relationship between the FieryDragonGameBoard and the Player class. I believe this will provide scalability as if there are more tokens or pseudo players added to the game. It can be added to this list of players in the FieryDragonGameBoard so that the game board notices its existence. Moreover, the utilisation of inheritance relationship between the Dragon Card abstract class and the PirateDragonCard and NormalDragonCard has also shown scalability. By using the abstract class we can then extend into more types of dragon card extension in the future. As it is easy to integrate new dragon cards since there is the abstract class which allows all new dragon cards to have the common foundation attributes and methods.

However, in the design rationale, it did not state the usage of some kind of factory class to generate the cards. This might incur some technical debt for future sprints . This might cause the software to be inflexible in creating the cards. Moreover it will make the creation of the card and the game to be tightly coupled which may cause issues when trying to update the card generation code. Hence, this inflexibility caused by no abstraction in the creation of the card objects will make the game rigid which is not good for usability purposes.

Maintainability

For the design rationale, it mentions the usage of inheritance to prevent code duplication and to further enhance extensibility of the application in the future. I support this statement as by creating abstract classes or interfaces we can ensure that the common aspects of different objects can be grouped up to form a foundational template that provides the basic structure of the concrete object. The example stated for the maintainability is true as well. There is the "DragonCard" abstract class that the PirateDragonCard and NormalDragonCard extends from show reusability as the methods like flipCard() and performAction(). This will then allow more different types of dragon card extension in the future.

Interaction Capability

The documentation of the design rationale cannot fully demonstrate the interaction of the game with the user so we cannot assess the interaction capability.

Nisha Kannapper

Functional Suitability

The design demonstrates a high level of functional suitability. It covers all essential functionalities, including initial board setup and player movement, with comprehensive methods and checks in place. The `movePlayer` method in the `Board` class ensures that player movements are validated correctly, considering various game rules such as move validity, player collisions, and completion of rounds. These thorough checks and validations ensure that the game operates correctly and efficiently, providing a smooth and accurate gameplay experience.

Flexibility

The design exhibits significant flexibility, primarily due to the use of inheritance and the creation of the abstract `Component` class. This setup allows for easy addition or modification of game components without requiring extensive changes to the existing codebase. Each component subclass inherits common attributes and methods from the `Component` class, making it straightforward to extend the game's functionality. The clear separation of responsibilities among classes, such as between `Game` and `GameController`, further enhances flexibility, allowing for modular and scalable development.

Maintainability

Maintainability is a strong aspect of the design, supported by its clear and organized structure. The use of the `Component` class to provide common functionality to its subclasses reduces code duplication and simplifies the overall codebase. By avoiding a "god class" through the creation of the `Game` class, the design ensures that responsibilities are well-distributed and that testing and maintenance are more manageable. Her structure makes it easier to understand, debug, and update the system, contributing to long-term maintainability.

Interaction Capability

The interaction capability is effectively addressed through the use of JavaFX for UI representation and well-defined methods for player interaction. The `initOnGrid` method ensures that each game component is correctly displayed on the grid, and any changes in component positions automatically update the UI. This results in a responsive and interactive user experience, allowing players to interact with the game seamlessly. The clear methods for handling player movements and game state updates ensure that the game responds accurately to player actions, enhancing the overall user engagement and interaction quality.

Tong Jet Kit

Functional Suitability

The design demonstrates a high level of functional suitability because it covers all basic requirements for the game, namely the board setup, and card flipping. There are distinct roles assigned to the Player and Card classes. The decision he made to separate the flipping action from the GameBoard and allocate it to the Player class shows the separation of functional responsibilities, which enhances the gameplay logic. This separation lets the game logic be somewhat centralised within appropriate classes, which avoids complexities compared with a "God" gameboard class. In the rationale, the key functionalities implemented have also been tested and the test cases pass so they are deemed to work properly, thus ensuring correctness as well. Besides that, completeness is also ensured as the game rules are adhered to with respect to the number of players being contained within the appropriate bounds, as well as the creation of volcano card groups of cells being combined together. Finally, the dragon cards are also randomised properly.

Flexibility

The design rationale demonstrates a moderate level of flexibility. The use of an abstract Card class as the base for the Volcano, Dragon and Cave Cards is a strong design choice that promotes flexibility. This abstraction lets us add new card types or modify existing ones without extensive changes to the overall codebase, which also directly increases the scalability of the system. However, the design rationale mentions hardcoded initialization for game cards, which could limit the system's ability to scale in the future. Since flexibility can be quantified by the number of hardcoded lines, this may affect the flexibility of the system. Reducing these hard coded elements could then improve the ability of the system to adapt to new requirements without an extensive reworking of the codebase.

Maintainability

The design rationale shows good maintainability because it uses abstract and inherited classes. The Card class serves as an abstract base for more specific card types like the DragonCards, VolcanoCards and CaveCards. These classes then have shared attributes and methods from the Card class. This approach will significantly reduce the effort required to introduce new card types or modify existing ones, as common features are centralized in the Card class. For example, if we add a new card called the "Ice Card", developers would simply just extend the abstract Card class, which ensures that all generic functionalities are inherited, while specific behaviours unique to the Ice Card can be coded in its own class. The metric in this case is that we can quantify the ease with which new card types can be added without modifying existing code, ideally aiming for minimal to no changes in the base Card class. Besides that, the implementation of the Flippable interface and FlipCardPane further enhances system flexibility. The Flippable interface lets different objects (like Cards) implement a standard set of methods related to the flipping action, which ensures that any object that should be flippable will adhere to this. This approach not only provides a clear abstraction but also lets us use polymorphic behaviour where different types of cards can exhibit their unique flipping animations or effects. The implementation of FlipCardPane encapsulates the functionality for displaying and animating the card flips, which separates the visual representation from the logic of the Cards themselves. This separation ensures that the UI changes can be made independently of the core game logic.

Interaction Capability

The interaction capabilities are also good based on the design rationale. By delegating the responsibility for the graphical representation and interaction handling to the controller class, this aligns with the Model-View-Controller (MVC) architectural pattern. This effectively separates the game logic from the concerns of the user interface. This ensures that user interactions such as clicking or flipping the cards are managed independently of game logic, which of course enhances the clarity of the user interface. For example, an effective interaction design is the method used for flipping cards. The controller handles the onClick events, which then triggers animations and subsequently calls logic-based methods in the Player class. This makes the game more responsive and also isolates the changes to the UI from game mechanics, which simplifies debugging end enhancements. The metric used is measured by the responsiveness of the UI, the intuitive nature of the controls, and the number and design of UI components in the game. This metric seems to have been met based on his design rationale.

Ong Chong How

Functional Suitability

Ong Chong How's design shows a relatively high level of functional suitability. This is exemplified by the deployment of the GameBoard and DragonCard classes. These classes encapsulate the critical functionalities of the game like handling the movement and setting up the board. For example, the GameBoard class manages the spatial layout and the player interactions which are part of the functional requirements of the game. For example, the GameBoard class manages the spatial layout and player interactions, whereas the DragonCard class handles the attributes and actions specific to the game, ensuring consistency. Besides that, other functionalities like card flipping and winning the game are also present, as well as the player movement on the board, so I can safely conclude that it is complete. In terms of the correctness, he shows that all test cases have passed, therefore we can say that it is correct as well for the functionalities that have been implemented by him. However, since some functionalities have not yet been implemented, we cannot say that it is 100% correct.

Since all scenarios for each implemented function have been tested, all test cases pass through a validation of game mechanics, it is decently correct and complete. Quantitatively, the number of basic classes implemented is more than 5, which covers all essential aspects of the game.

Flexibility

The design shows a moderate level of flexibility, because there is a presence of an abstract "DragonCard" class, which serves as a base class for deriving specialised card types as mentioned in his rationale, like "AnimalDragonCard" and "PirateDragonCard". This hierarchical class structure lets us easily extend the code and integrate new card functionalities without altering the game logic fundamentally.

Besides that, according to the design rationale, there is also a relatively low number of hardcoded lines, and it uses dynamic code techniques to manage game elements. The value here would be the number of hardcoded lines, and since there is a minimal presence of hardcoded lines, we can safely consider this design to be somewhat flexible as developers can scale the application easily without hardcoding more lines.

Maintainability

The design is relatively maintainable as well, as per earlier with the abstract DragonCard class that manages common card properties and behaviours expected of a typical generalised DragonCard. This will then effectively reduce the redundancy and repeated code, and it follows the Don't Repeat Yourself (DRY) principle so we don't have to code up the same functionalities again if we make a new card and instead just inherit the common functionalities when making a new type of DragonCard.

The metric here is that the maintainability is reflected by the use of abstract classes, which streamline the addition of new card types and modifications of existing functionalities. The value is the number of abstract classes.

<u>Interaction Capability</u>
The interaction capabilities in the rationale are also quite robust, as there are a number of interactive UI components managed by the game's controller class. This separation of UI logic in controller classes also follows the Model-View Controller (MVC) method, which allows developers in the future to work on the UI without interfering with game logic.

The metric here is that interaction capability is enhanced by UI components that respond to user's actions, like card flips and movement of the player, and it provides immediate visual feedback to maintain player engagement. This is all present in the design rationale as there are interactive components in the controller to update the UI like buttons. The value would then be the number of buttons and listeners for gesture detections.

Outcome of the Assessment

In Sprint 2, our team focused on implementing various functionalities of the game, with each member contributing significantly to different aspects. Our team has made strategic decisions regarding the adoption of specific code implementations to enhance the game's functionality and user experience. The chosen implementations for the UI, card flipping, player movement, and turn-changing functionalities have been carefully selected based on their completeness and visual appeal. This report outlines the decisions made regarding the adoption of specific code implementations, the rationale behind these choices, and potential improvements for the next sprint.

Firstly, for the UI and gameboard object creation, we decided to utilise Tong Jet Kit's code implementation to create them. This is because Tong Jet Kit's UI fits the game requirement effectively. Additionally, the card combinations in his implementation align perfectly with the basic game setup. Moreover, he also implemented the card randomization which is correct and complete as well. Furthermore, Jet Kit's placement of the dragon cards adds an interesting dynamic to the gameplay. Hence we have chosen to use Jet Kit's code implementation on setting up the board due to its alignment with the game's requirement.

Secondly, for the Flipping the Dragon Card functionality, our team has unanimously decided to utilise Tong Jet Kit's implementation. Despite Chong How's contribution in this key game functionality through Python, the absence of an unflip method made it challenging to verify correctness. In Jet Kit's implementation, the flipping functionality also includes a visually appealing animation for flipping cards. Hence, we have decided to adopt Tong Jet Kit's implementation for flipping dragon cards due to its completeness and added visual effects.

Thirdly, for the Movement of Player Token based on the Flipped Dragon Card, Mandhiren and Nisha have contributed in this game functionality. Mandhiren's implementation on the movement is accurate and comprehensive which includes handling moving forward when selecting the correct dragon card with the right creature and moving backwards when selecting the pirate dragon card. Nisha's implementation is also correct and complete where the player can move forward and backwards to the correct position. However, Nisha implementation has an extra functionality where the player will go back to the cave after traversing the board for 1 rotation. Both of them have also implemented the correct and complete functionality for this key game functionality. Since it is like this, We decided to use both Mandhiren's and Nisha's implementations as references for integrating this key functionality, ensuring a robust and error-free movement mechanism.

Next, we have the Changing Turn to Next Player implementation. For this key game functionality, we decided to use Mandhiren's code implementation for it. This is because for his implementation there was a show on how the turn is changed through a gold ring indicator, which visually marks the active player. With this, it can clearly demonstrate how the turn changes from 1 player to another. Thus, Mandhiren's Changing Turn to Next Player codebase will be utilised in the Sprint 3 tech prototype due to its clarity and effective visual representation of the active player.

Lastly, for the win game implementation. Since our group has only 3 members in Sprint 2, none of our team members implement the win game functionality as we have chosen to do the Flipping the Dragon Card, Movement of Player Token and Changing Turn to Next Player. Thus, the Winning The Game functionality will be implemented in Sprint 3 with no ideas getting from Sprint 2 tech prototypes.

For new ideas to improve the prototype, our proposed improved ideas for the prototype is to adjust the size of the screen so that it's not too big until it covers the whole screen for some display resolution. This adjustment is to ensure the game screen size will be adequate for different monitor sizes. Moreover, we also decided to add an exit button on the game screen so it can exit to the main menu which is the start screen. This is so that the user can exit to the main menu and start a new game quickly without needing to close the game and then reopen the game again. These quality of life changes will then improve the user experience when playing the game.

CRC: FieryDragonsGameBoard

FieryDragonsGameBoard	
Responsibility	**Collaborators**
Initialise the game board layout Add volcano cards to the board Add caves to the board Add dragon cards to the board Initialise players Determine destination based on player's move Manage current player Provide access to game elements Manage game board state	VolcanoCard LocationCard CaveCard DragonCard Player

Justification

The `FieryDragonsGameBoard` class is like the main organiser of the Fiery Dragons game. It's in charge of representing the physical game board. Here's a breakdown of what it does and who it works with:

1. Setting Up the Game Board:

This class starts by getting the game board ready. It's like setting up the stage before a play begins. It prepares the layout and gets everything ready for the game elements to be placed.

2. Adding Game Elements:

Once the board is ready, the class works with other parts of the game to put things like volcano cards, cave cards, and dragon cards onto the board. These cards add challenges, goals, and surprises to the game.

3. Getting Players Ready:

Before the game starts, players need to be set up. This class handles that job, making sure each player has what they need to begin their journey.

4. Managing Player Turns:

During the game, players take turns moving around the board. This class keeps track of whose turn it is and helps players figure out where they can go next.

5. Giving Access to Game Elements:

Players need to interact with different parts of the game, like picking up cards or moving to new locations. This class lets them do that by giving them access to all the game elements they need.

6. Keeping Track of the Game Board:

Throughout the game, the class keeps an eye on what's happening on the board. It makes sure to update everyone on any changes, like when players move or when new cards are placed.

Overall, the `FieryDragonsGameBoard` class is like the behind-the-scenes manager of the game, making sure everything works smoothly so that players can have fun and enjoy the game. There was an alternative decision of giving the FieryDragonsGameBoard the responsibility to create the UI. However we decided to delegate this task to the FieryDragonGameBoardController class as that class is more suitable to handle jobs related to the UI.

CRC: FieryDragonGameBoardController

FieryDragonsGameBoardController	
Responsibility	**Collaborators**
Initialise the game board Set up the UI for the game board elements Place player tokens on the board Handle click events on dragon cards End the current player's turn Change to the next player's turn Display winning message Create information cards Style labels	FieryDragonsGameBoard VolcanoCardGroup VolcanoCard CaveCard DragonCard Player FlipCardPane

Justification

The `FieryDragonGameBoardController` class serves as the central hub for controlling the user interface and managing game logic in the Fiery Dragons game. Its primary purpose is to mediate between the graphical representation of the game board and the underlying game mechanics, ensuring a seamless gaming experience for the players.

At its core, this controller is responsible for initialising the game board upon startup. This involves creating the game board object. Through its `initialize()` method, it orchestrates the setup process, ensuring that all necessary components are created and in place before gameplay begins.

Once initialised, the controller handles the setup of the UI elements through the `setUpUI()` method. This includes creating and positioning the volcano, cave, and dragon cards on the game board UI, as well as placing player tokens on their starting positions. By encapsulating this functionality within the controller, it maintains a clear separation of concerns and promotes code readability.

During gameplay, the controller responds to user interactions, particularly mouse clicks on dragon cards. It implements the `setDragonCardClickListener()` method to handle these clicks events, allowing players to flip dragon cards and reveal their identities. Additionally, it manages the end of each player's turn through the `endTurn()` method, ensuring that flipped cards are reset and control is passed to the next player.

Furthermore, the controller handles the display of winning messages through the `displayWinningMessage()` method. This functionality is triggered when a player successfully completes the game objectives, signalling the end of the game and congratulating the winning player.

Overall, the `FieryDragonGameBoardController` class plays a crucial role in orchestrating the various components of the Fiery Dragons game, from setup to gameplay to conclusion. By encapsulating these responsibilities within a single class, it promotes code organisation, modularity, and maintainability.

Alternative distributions of responsibilities for this class were considered but ultimately rejected due to increased complexity and potential issues with cohesion and coordination. One alternative was to split UI setup and game logic into two controllers, which was dismissed because it would complicate communication and create redundancy. Another idea was to delegate responsibilities to individual components, but this was rejected due to risks of inconsistent game state management and difficulties in coordination. A third option involved creating sub-controllers for specific tasks, which would add overhead and fragment responsibilities without significant benefits. The current design, which centralises these responsibilities in a single controller, ensures high cohesion and simplifies the development and maintenance of the game, making it the preferred approach.

CRC: Player

Player	
Responsibility	**Collaborators**
Track player's current location Move player on the game board Generate player's token Highlight player's token Unhighlight player's token Update player's move count Set its attributes Get its attributes	LocationCard GridPane CreatureType VolcanoCard

Justification

The `Player` class functions as a central entity within the game architecture, embodying the attributes and behaviours of individual players within the game. Its primary responsibility lies in managing the state of each player, including their current location on the game board, denoted by a specific card or position, their respective tokens and their progress towards winning the game. This includes tracking the number of moves made by each player, which is crucial for determining game progression and victory conditions. Additionally, the class handles the visual representation of players on the game board through tokens, facilitating easy identification and interaction during gameplay.

One of the core purposes of the `Player` class is to enable player actions and interactions within the game environment. This includes handling movement across the game board, which involves updating the player's current location based on their movement choices and the rules of the game. By encapsulating movement logic within the `Player` class, the system ensures a consistent and reliable mechanism for player navigation, enhancing the overall gameplay experience.

Moreover, the `Player` class plays a crucial role in determining game outcomes, particularly in assessing whether a player has achieved victory conditions. This involves monitoring the player's progress, typically represented by reaching a certain location or completing specific objectives within the game. By encapsulating win conditions and victory determination within the `Player` class, the system maintains a clear and modular structure, making it easier to implement and adjust game rules and objectives.

Additionally, the `Player` class facilitates interaction with other game components, such as the game board and other players. This includes handling player highlights or indicators to signify active turns, managing player interactions during gameplay events, and coordinating player-related actions with other game elements.

Alternatively, the player class initially does not have the movement responsibility, but is delegated to the gameboard. This is because, before this decision, the player is just a data holder class. After further discussion, we believe it is more suitable to give the player the movement responsibility as the player should be the one moving the tokens. Hence it should be the player's responsibility rather than the gameboard.

CRC: Card

Card	
Responsibility	**Collaborators**
Represent a card in the game Hold the symbol of the card Hold the image view of the card Hold the type of the card Set the card type Set the card symbol Set the card image view Get the card type Get the card symbol Get the card image view	Symbol CardType

Justification

The `Card` class forms the foundational structure for representing individual cards within the game framework, encapsulating key attributes and functionalities essential for card-based gameplay. It embodies the central concept of cards in the game context, serving as a blueprint from which specific card instances can be derived. At its core, this class is designed to provide a unified interface for handling various types of cards, regardless of their specific properties or behaviours. By abstracting common characteristics such as symbol, image view, and type, the `Card` class establishes a standardised template that promotes consistency and reusability across different card types.

One of the primary purposes of the `Card` class is to encapsulate essential card-related information and functionalities within a single entity. By holding attributes such as the card symbol, image view, and type, the class effectively captures the intrinsic characteristics of each card instance. This facilitates seamless integration with other game components and simplifies the management of card-related operations throughout the game lifecycle. Additionally, the class provides methods for setting and retrieving card attributes, enabling dynamic manipulation and interaction with card objects during gameplay.

Furthermore, the `Card` class serves as a foundation for implementing diverse card-based mechanics and features within the game system. Its abstract nature allows for the creation of specialised card subclasses tailored to specific gameplay requirements, such as dragon cards, volcano cards, or treasure cards. Each subclass can extend the base functionality of the `Card` class to incorporate unique behaviours and properties associated with different card types. This modular approach not only enhances code maintainability and extensibility but also promotes flexibility in adapting to evolving game design needs.

Moreover, the `Card` class fosters a separation of concerns by encapsulating card-related logic within a cohesive entity, distinct from other game components. This architectural principle promotes code clarity and modularity, making it easier to manage and reason about card-related functionalities independently of other game mechanics. By abstracting away implementation details behind a unified interface, the class promotes code reuse and facilitates the development of scalable and maintainable game systems. Overall, the `Card` class plays a foundational role in defining and orchestrating card-based interactions within the broader context of the game environment.

Alternative distributions of responsibilities for the 'Card' class were considered but ultimately rejected due to various issues. One proposal was to integrate card-specific game logic within the 'Card' class, which was dismissed because it would increase complexity, reduce reusability, and violate the single responsibility principle. Another idea was to create specialised subclasses for each card type, such as 'DragonCard' and 'VolcanoCard', but this would lead to code duplication, increased maintenance effort, and reduced flexibility. A third option involved externalising attribute management to a 'CardAttributeManager' class, which would add unnecessary indirection, introduce performance overhead, and decrease cohesion. The current design, where the 'Card' class encapsulates the representation and management of card attributes, ensures simplicity, maintainability, and adherence to object-oriented principles, making it the preferred approach.

CRC: DragonCard

DragonCard	
Responsibility	**Collaborators**
Represent a dragon card in the game Be flipped by a player Manage the number of creatures on the card	Card FieryDragonsGameBoard Player

Justification

The `DragonCard` class serves as a crucial component within the Fiery Dragons game architecture, embodying the attributes and functionalities specific to dragon cards. These Dragon Cards are cards that the player will use to determine its movement. Primarily, it encapsulates the representation and behavior of dragon cards in the game, extending the generic `Card` class and implementing the `Flippable` interface. This design choice allows for a modular and scalable approach to managing different types of cards within the game, ensuring consistency and flexibility in card-related operations.

One of the core responsibilities of the `DragonCard` class is to manage the flipping mechanism associated with dragon cards during gameplay. Through the `flipCard` method, the class determines the outcome of flipping a dragon card based on the current game state and player actions. This includes evaluating whether the flipped card matches certain criteria, such as having the same symbol as the next location card or being a special type of dragon card, such as a Pirate Dragon Card. By encapsulating flipping logic within the `DragonCard` class, the system maintains cohesion and clarity in handling card-related interactions, enhancing gameplay dynamics and immersion.

Moreover, the `DragonCard` class facilitates interactions between dragon cards and other game elements, such as players and the game board. For instance, it helps players to update its positions based on the outcome of flipping a dragon card, potentially triggering movement or game events based on specific card attributes or conditions. Additionally, the class collaborates with the `FieryDragonsGameBoard` and `Player` classes to manage game state and player actions, ensuring synchronization and consistency across different aspects of gameplay. This cohesive integration of card behavior with broader game mechanics contributes to a seamless and engaging gaming experience for players.

Furthermore, the `DragonCard` class encapsulates information about the number of creatures present on each dragon card. This attribute plays a crucial role in determining gameplay outcomes, as it may influence player movement, game events, or victory conditions. This is because the number of creature count will determine the number of steps a player can move. By encapsulating creature count management within the `DragonCard` class, the system maintains a clear and structured representation of card attributes, facilitating easier manipulation and interpretation of card-related data. This abstraction enhances the modularity and extensibility of the game system, allowing for easier integration of new card types or mechanics in future updates or expansions.

Alternative distributions of responsibilities for the 'DragonCard' class were considered but ultimately rejected due to various drawbacks. One proposal was to merge the flipping mechanism and creature count management into the broader 'Card' class or an external manager, but this was dismissed because it would increase the complexity, reduce the clarity of class responsibilities, and violate the single responsibility principle. Another idea was to distribute the flipping logic and creature count handling to the 'FieryDragonsGameBoard' and 'Player' classes, which would fragment the core functionality of the 'DragonCard' and lead to a less cohesive design. This fragmentation would complicate the synchronisation and consistency of game state management, increasing the risk of bugs and making the system harder to maintain. The current design, where the DragonCard class extends the generic Card class and implements the Flippable interface, ensures modularity, cohesion, and clear encapsulation of dragon card-specific behaviours, making it the most effective approach for maintaining a scalable and maintainable game architecture.

CRC: LocationCard

LocationCard	
Responsibility	**Collaborators**
Represent a location card in the game Manage spatial attributes (row, column) Establish linked relationships with other cards Determine player occupancy Determine if it has a card linked to it.	Card FieryDragonsGameBoard Player

Justification

The `LocationCard` class serves as a fundamental component within the Fiery Dragons game architecture, responsible for representing the various locations where players can reside during gameplay. Its primary role revolves around managing the spatial attributes and characteristics of these location cards within the game board. This includes storing information such as the row and column indices of each card on the game board grid, essential for determining the layout and positioning of cards during gameplay.

A key aspect of the `LocationCard` class is its ability to establish linked relationships with other cards within the game environment. This functionality enables the creation of connections or pathways between different locations on the game board, allowing players to navigate between various areas during gameplay. For example, between the cave cards and the volcano cards. By providing methods for setting and retrieving linked cards, the `LocationCard` class facilitates dynamic interactions and movement possibilities within the game, contributing to the overall depth and complexity of gameplay mechanics.

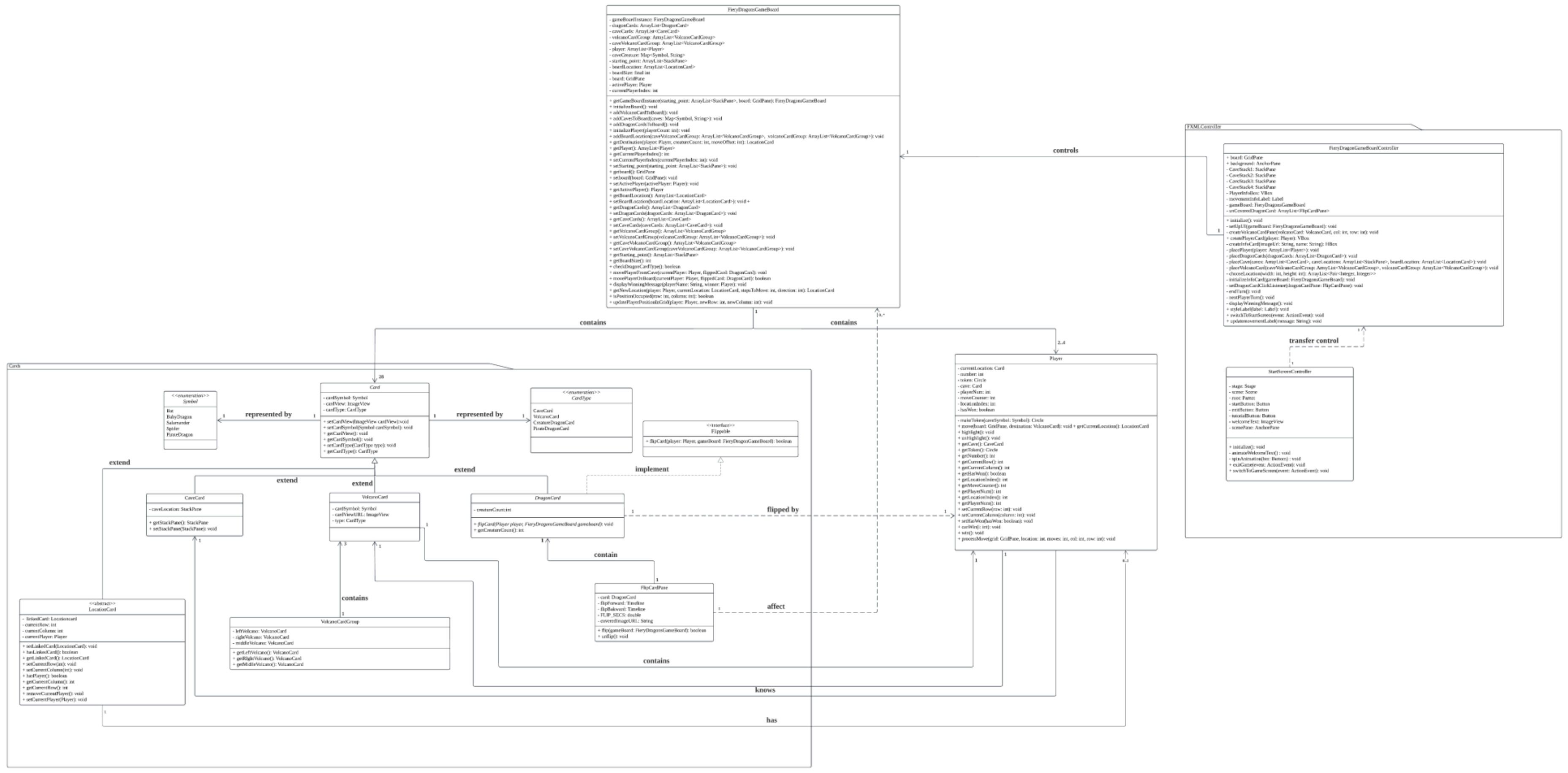
Furthermore, the `LocationCard` class plays a crucial role in managing player occupancy within the game environment. Through methods such as `hasPlayer()`, the class determines whether a player is currently occupying a specific location card on the game board. This functionality is essential for enforcing game rules related to player movement and interaction. This is because in the game only 1 player can occupy a volcano card tile.

Additionally, the `LocationCard` class encapsulates essential attributes and behaviours common to all location cards in the game. By extending the `Card` superclass, it inherits functionalities related to card symbols, types, and image representations, providing a consistent interface for interacting with location cards throughout the game. This design approach promotes code reusability and modularity, simplifying the implementation and maintenance of location-based game features within the Fiery Dragons game system.

Alternative distributions of responsibilities for the ‘LocationCard’ class were considered but ultimately rejected due to several issues. One proposal was to integrate spatial attributes and linked relationships into the broader ‘FieryDragonGameBoard’ class. However, this approach was dismissed because it would centralise too many responsibilities in the game board, increasing its complexity and reducing the modularity and clarity of individual components. Another idea involved distributing the responsibilities of managing player occupancy and linked relationships to the ‘Player’ and ‘Card’ classes. This was also rejected, as it would lead to fragmented logic and reduced cohesion, making it difficult to maintain and extend the game. The current design, where the ‘LocationCard’ class manages its spatial attributes, linked relationships, and player occupancy while extending the generic ‘Card’ class, ensures a clear and modular structure. This design promotes code reusability, maintainability, and a cohesive representation of location-specific behaviours, providing a robust foundation for the game’s architecture.

UML Diagram

Lucid Chart Link: https://lucid.app/lucidchart/5f53778e-44a8-43c5-bacf-0101f3127e71/edit?view_items=_QgBuRMIe5qJ&invitationId=inv_4e782b54-80fa-4066-83c8-077067e3a230



FieryDragonsGameBoard
- gameBoardInstance: FieryDragonsGameBoard
- dragonCards: ArrayList<DragonCard>
- caveCards: ArrayList<CaveCard>
- volcanoCardGroup: ArrayList<VolcanoCardGroup>
- caveVolcanoCardGroup: ArrayList<VolcanoCardGroup>
- player: ArrayList<Player>
- caveCreature: Map<Symbol, String>
- starting_point: ArrayList<StackPane>
- boardLocation: ArrayList<LocationCard>
- boardSize: final int
- board: GridPane
- activePlayer: Player
- currentPlayerIndex: int
+ getGameBoardInstance(starting_point: ArrayList<StackPane>, board: GridPane): FieryDragonsGameBoard
+ initializeBoard(): void
+ addVolcanoCardToBoard(): void
+ addCavesToBoard(caves: Map<Symbol, String>): void
+ addDragonCardsToBoard(): void
+ initializePlayer(playerCount: int): void
+ addBoardLocation(caveVolcanoCardGroup: ArrayList<VolcanoCardGroup>, volcanoCardGroup: ArrayList<VolcanoCardGroup>): void
+ getDestination(player: Player, creatureCount: int, moveOffset: int): LocationCard
+ getPlayer(): ArrayList<Player>
+ getCurrentPlayerIndex(): int
+ setCurrentPlayerIndex(currentPlayerIndex: int): void
+ setStarting_point(starting_point: ArrayList<StackPane>): void
+ getboard(): GridPane
+ setboard(board: GridPane): void
+ setActivePlayer(activePlayer: Player): void
+ getActivePlayer(): Player
+ getBoardLocation(): ArrayList<LocationCard>
+ setBoardLocation(boardLocation: ArrayList<LocationCard>): void +
+ getDragonCards(): ArrayList<DragonCard>
+ setDragonCards(dragonCards: ArrayList<DragonCard>): void
+ getCaveCards(): ArrayList<CaveCard>
+ setCaveCards(caveCards: ArrayList<CaveCard>): void
+ getVolcanoCardGroup(): ArrayList<VolcanoCardGroup>
+ setVolcanoCardGroup(volcanoCardGroup: ArrayList<VolcanoCardGroup>): void
+ getCaveVolcanoCardGroup(): ArrayList<VolcanoCardGroup>
+ setCaveVolcanoCardGroup(caveVolcanoCardGroup: ArrayList<VolcanoCardGroup>): void
+ getStarting_point(): ArrayList<StackPane>
+ getBoardSize(): int
+ checkDragonCardType(): boolean
+ movePlayerFromCave(currentPlayer: Player, flippedCard: DragonCard): void
+ movePlayerOnBoard(currentPlayer: Player, flippedCard: DragonCard): boolean
+ displayWinningMessage(playerName: String, winner: Player): void
+ getNewLocation(player: Player, currentLocation: LocationCard, stepsToMove: int, direction: int): LocationCard
+ isPositionOccupied(row: int, column: int): boolean
+ updatePlayerPositionInGrid(player: Player, newRow: int, newColumn: int): void
controls
1
1
contains
contains
1
0..*
28
2..4
FXMLController
FieryDragonGameBoardController
+ board: GridPane
+ background: AnchorPane
- CaveStack1: StackPane
- CaveStack2: StackPane
- CaveStack3: StackPane
- CaveStack4: StackPane
- PlayerInfoBox: VBox
- movementInfoLabel: Label
- gameBoard: FieryDragonsGameBoard
- unCoveredDragonCard: ArrayList<FlipCardPane>
+ initialize(): void
- setUpUI(gameBoard: FieryDragonsGameBoard): void
- createVolcanoCardPane(volcanoCard: VolcanoCard, col: int, row: int): void
+ createPlayerCard(player: Player): VBox
- createInfoCard(imageUrl: String, name: String): HBox
- placePlayer(player: ArrayList<Player>): void
- placeDragonCards(dragonCards: ArrayList<DragonCard>): void
- placeCave(caves: ArrayList<CaveCard>, caveLocations: ArrayList<StackPane>, boardLocation: ArrayList<LocationCard>): void
- placeVolcanoCard(caveVolcanoCardGroup: ArrayList<VolcanoCardGroup>, volcanoCardGroup: ArrayList<VolcanoCardGroup>): void
- chooseLocation(width: int, height: int): ArrayList<Pair<Integer, Integer>>
- initializeInfoCard(gameBoard: FieryDragonsGameBoard): void
- setDragonCardClickListener(dragonCardPane: FlipCardPane): void
- endTurn(): void
- nextPlayerTurn(): void
- displayWinningMessage(): void
+ styleLabel(label: Label): void
+ switchToStartScreen(event: ActionEvent): void
+ updatemovementLabel(message: String): void
transfer control
1
1
StartScreenController
- stage: Stage
- scene: Scene
- root: Parent
- startButton: Button
- exitButton: Button
- tutorialButton: Button
- welcomeText: ImageView
- scenePane: AnchorPane
+ initialize(): void
- animateWelcomeText() : void
- spinAnimation(btn: Button) : void
+ exitGame(event: ActionEvent): void
+ switchToGameScreen(event: ActionEvent): void
Cards
<<enumeration>>
Symbol
Bat
BabyDragon
Salamander
Spider
PirateDragon
represented by
1
1
Card
- cardSymbol: Symbol
- cardView: ImageView
- cardType: CardType
+ setCardView(ImageView cardView):void
+ setCardSymbol(Symbol cardSymbol): void
+ getCardView(): void
+ getCardSymbol(): void
+ setCardType(CardType type): void
+ getCardType(): CardType
represented by
1
1
<<enumeration>>
CardType
CaveCard
VolcanoCard
CreatureDragonCard
PirateDragonCard
<<interface>>
Flippable
+ flipCard(player: Player, gameBoard: FieryDragonsGameBoard): boolean
extend
extend
extend
extend
implement
CaveCard
- caveLocation: StackPane
+ getStackPane(): StackPane
+ setStackPane(StackPane): void
1
VolcanoCard
- cardSymbol: Symbol
- cardViewURL: ImageView
- type: CardType
1
3
1
DragonCard
- creatureCount:int
+ flipCard(Player player, FieryDragonsGameBoard gameboard): void
+ getCreatureCount(): int
1
flipped by
1
1
contain
1
FlipCardPane
- card: DragonCard
- flipForward: Timeline
- flipBakward: Timeline
- FLIP_SECS: double
- coveredImageURL: String
+ flip(gameBoard: FieryDragonsGameBoard): boolean
+ unflip(): void
1
affect
contains
1
<<abstract>>
LocationCard
- linkedCard: Locationcard
- currentRow: int
- currentColumn: int
- currentPlayer: Player
+ setLinkedCard(LocationCard): void
+ hasLinkedCard(): boolean
+ getLinkedCard(): LocationCard
+ setCurrentRow(int): void
+ setCurrentColumn(int): void
+ hasPlayer(): boolean
+ getCurrentColumn(): int
+ getCurrentRow(): int
+ removeCurrentPlayer(): void
+ setCurrentPlayer(Player): void
VolcanoCardGroup
- leftVolcano: VolcanoCard
- rightVolcano: VolcanoCard
- middleVolcano: VolcanoCard
+ getLeftVolcano(): VolcanoCard
+ getRightVolcano(): VolcanoCard
+ getMiddleVolcano(): VolcanoCard
contains
knows
1
has
Player
- currentLocation: Card
- number: int
- token: Circle
- cave: Card
- playerNum: int
- moveCounter: int
- locationIndex: int
- hasWon: boolean
- makeToken(caveSymbol: Symbol): Circle
+ move(board: GridPane, destination: VolcanoCard): void + getCurrentLocation(): LocationCard
+ highlight(): void
+ unHighlight(): void
+ getCave(): CaveCard
+ getToken(): Circle
+ getNumber(): int
+ getCurrentRow(): int
+ getCurrentColumn(): int
+ getHasWon(): boolean
+ getLocationIndex(): int
+ getMoveCounter(): int
+ getPlayerNum(): int
+ getLocationIndex(): int
+ getPlayerNum(): int
+ setCurrentRow(row: int): void
+ setCurrentColumn(column: int): void
+ setHasWon(hasWon: boolean): void
+ canWin(): int): void
+ win(): void
+ processMove(grid: GridPane, location: int, moves: int, col: int, row: int): void
1
1
0..1

Description of your executable

The executable for our project is a .jar file since our project is created using Java.

Platform

Programming Language	Java
Operating System	Windows, MacOs, Linux
Development Toolkit	JDK 22
UI Library	JavaFx 22.0.1 SDK

Instructions to Run the Game

1. Make sure you have the JDK 22 Development Kit 22: https://www.oracle.com/my/java/technologies/downloads/#jdk22-windows

2. Download the FieryDragonsGame.jar FILE

3. Run the jar file using Java(TM) Platform SE binary.

Building Instructions

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 the 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 called “out”.