Assessment Criteria Definitions (including the corresponding metrics how each of these criteria is to be assessed)

Completeness of the solution direction (i.e. to what extent are all key functionalities covered in the design) [Functional completeness; functional correctness]

- Functional correctness - Absolute
    - Based on the key functionalities stated in Sprint 2:
        - set up the initial game board (including randomised positioning for dragon cards)
        - flipping of dragon ("chit") cards
        - movement of dragon tokens based on their current position as well as the last flipped dragon 1 card
        - change of turn to the next player
        - winning the game.

      How many of these key functionalities are implemented correctly
- Functional completeness - Ordinal
    - Degree to which set of functions covers all specified tasks (How much functionality the defined functions can cover)
        - Covers none - The defined functions cannot be used to cover any required functionality
        - Covers some - The defined functions can be used to cover some of the required functionality
        - Covers most - The defined functions can be used to cover most of the required functionality
        - Covers all - The defined functions can be used to cover all of the required functionality

Rationale behind the chosen solution direction [Functional appropriateness]

- Ordinal scale
    - How well the functions facilitate the execution of user stories/game rules, e.g., design patterns.
        - Not appropriate - the design of the system is completely unable to facilitate the functionality needed for the system
        - Somewhat appropriate - the design of the system is able to facilitate some of the functionality needed for the system
        - Very appropriate - the design of the system is able to facilitate most of the functionality needed for the system
        - Extremely appropriate - the design of the system is able to facilitate all if not all of the functionality needed for the system

Understandability of the solution direction [Appropriateness recognizability]

- Ordinal scale
    - How understandable is the facilitation of the execution of key game features; is rationale overly complex, or easy to grasp and work with
        - Not understandable - the rationale behind the design of the system is unable to be understood, or would require a disproportionate amount of time to grasp knowledge of implementation

        - Somewhat understandable - the rationale behind the design of the system is somewhat able to be understood and requires slightly more than expected amount of time to grasp knowledge of implementation
        - Very understandable - the rationale behind the design of the system is able to be understood quite easily and requires the expected amount of time to grasp knowledge of implementation
        - Extremely understandable - the rationale behind the design of the system is able to be understood extremely easily and requires much less time than expected to grasp knowledge of implementation

Extensibility of the solution direction (in anticipation of extensions to the game for Sprint 4) [Modifiability];

- Ordinal scale
    - How modifiable and extendable the solution is with regards to potential extensions to the game
        - Not extensible - The solution is not extendable, would have to change overall structure
        - Somewhat extensible - The solution is somewhat extensible, would have to change some structure
        - Very extensible - The solution is mostly extensible, would have to make minor changes to structure
        - Extremely extensible - The solution is completely extensible, no structural changes required

Quality of the written source code (e.g., coding standards, reliance on case analysis and/or down-casts) [Maintainability]; https://google.github.io/styleguide/javaguide.html

- Absolute + Ordinal
    - With regards to reliance on case analysis and or downcasts, it can be counted
    - With regards to coding standards, an ordinal scale is used for measure
        - Doesn't conform - the coding standards used does not conform to Google Java Coding Guidelines
        - Somewhat conforms - the coding standards used somewhat conform to Google Java Coding Guidelines
        - Mostly conforms - the coding standards used mostly conforms to guidelines as specified by the Google Java Style Guide
        - Conforms - the coding standards used completely conforms to guidelines as specified by the Google Java Style Guide

Aesthetics of the user interface [User engagement];

- Ordinal
    - How likely is the user interface to encourage continued user interaction (considering given functions and information)
        - Not likely - User interface is not visually appealing, user would not want to continue playing
        - Somewhat likely - User interface is somewhat visually appealing, user would continue playing
        - Very likely - User interface is visually appealing, user would like to continue playing

Usability [Operability, self descriptiveness, user assistance]:

- Ordinal
    - How easy the system is to operate and control, and how understandable and accommodating the system is
        - Not understandable - User does not know how to or cannot navigate the game
        - Somewhat understandable - The user can navigate the game with difficulty
        - Mostly understandable - The user can navigate the game with little difficulty
        - Extremely understandable - The user can navigate the game intuitively

Review of Prototypes

Group Member 1: Krishna

Completeness of the solution direction (i.e. to what extent are all key functionalities covered in the design) [Functional completeness; functional correctness]

- Functional Correctness: 4
  Does not account for edge cases when moving the player. Including checking if the player can move when they are moving past the cave and winning the game.
- Functional Completeness: 5
  All core game features are implemented and designed.

Rationale behind the chosen solution direction [Functional appropriateness]

- Very appropriate
  Covers all functionality, and each class has its own purpose without obvious god classes. Deck class currently provides no value and purpose. After clarification, it is implemented with future extensions in mind. No design pattern was explicitly followed/utilised in this context which may introduce a few problems that are not apparent as of now.

Understandability of the solution direction [Appropriateness recognizability]

- Somewhat understandable
  Some features of the game can be difficult to understand at times, requiring a deeper look at the sequence diagrams.

Extensibility of the solution direction (in anticipation of extensions to the game for Sprint 4) [Modifiability];

- Very extensible
  Various classes such as deck class, MapPiece and NormalPiece allow for various extensions with regards to how the chitcards and the game board can be modified. All extensions with regards to UI/UX can be extended through the JavaFX controllers. Extensions dealing with turns and modification of turns can also be implemented by potentially modifying the TurnManager class.

Quality of the written source code (e.g., coding standards, reliance on case analysis and/or down-casts) [Maintainability];

- Many coding standards were followed. Nothing obvious can be pointed out.

- Very little case analysis and no downcasts were used.
- Repetition can be found throughout the code especially in the controller classes. This can be improved.

Aesthetics of the user interface [User engagement];

- The user interface is a bit lacking. The overall design of the application and the game board is simple rectangles rather than circular.

Usability [Operability, self descriptiveness, user assistance]:

- Very small number of buttons for control, easily guides the user how to navigate and play the game.
- Information is very clearly presented without excessive interactions. The user is able to navigate and use the application without queries.
- Little to none user assistance is provided.

Summary of key findings:

- The key features of the Fiery Dragons board game are implemented and designed to a high level. However when implementing moving the player, the design has not considered checking for moving past a cave that the player will move to when winning the game. No apparent design pattern was utilised, and a few classes appear to have functions with the context of the base Fiery Dragons game. These classes are implemented with respect to future extensions that may be added to the game.

Group Member 2: Jeffrey KRISHNA DO

Completeness of the solution direction (i.e. to what extent are all key functionalities covered in the design) [Functional completeness; functional correctness]

- Functional Correctness: 5
  Includes all required features
- Functional Completeness: 5
  All functions can be used to implement required features

Rationale behind the chosen solution direction [Functional appropriateness]

- Very appropriate
  - Good logic behind functions
  - All classes have a purpose and no unnecessary classes
  - Deck class currently provides no value, however prepares for later extension
  - Appropriate use of static class for Turn

Understandability of the solution direction [Appropriateness recognizability

- Very understandable
  - Chosen approach is easy to understand
  - Sequence diagrams are logical and provide insight into chosen approach

Extensibility of the solution direction (in anticipation of extensions to the game for Sprint 4) [Modifiability];

- Very extensible

    - Extensible board design by utilising Tiles and MapPieces
    - Extensible deck design, while the Deck class currently serves no purpose, it can be used later for game extensions

Quality of the written source code (e.g., coding standards, reliance on case analysis and/or down-casts) [Maintainability];

- No obvious deviations from coding standards
- Appropriate class and function names
- In line comments used where necessary
- Could include Javadoc for more complex classes
- Some case analysis used in controller class, however this may be necessary and unavoidable

Aesthetics of the user interface [User engagement];

- Somewhat likely
    - UI representation of game is similar to original board game design
    - Could include animal designs for dragon cards and tiles to more accurately reflect board game
    - Board tiles contain various shades rather than a single block colour for the entire board
    - Easily able to identify which players turn it is

Usability [Operability, self descriptiveness, user assistance]:

- Somewhat understandable
    - Using the initial landing page is clear and easy to understand, slider is intuitive and start game button is clearly labelled
    - Assumes knowledge or how Fiery Dragons board game works to play, clicking on cards to progress with turns
    - No user assistance/game instructions are provided

Summary of key findings:

- Good UI design, having a clear circular board with various coloured tiles, however could add images instead of words to increase user engagement
- Extensible design, should make use of MapPiece and Tile, and DragonCard and Deck classes in Sprint 3 implementation
- While design is visually appealing, need to include game instructions in Sprint 3 implementation for new players
- Ensure that complex methods in Sprint 3 implementation have Javadoc

Group Member 3: Zilei

Completeness of the solution direction (i.e. to what extent are all key functionalities covered in the design) [Functional completeness; functional correctness]

- Functional Correctness: 5
- Functional Completeness: 5

Rationale behind the chosen solution direction [Functional appropriateness]

- Very appropriate
    - Use of Board as singleton is intuitive and implementation is appropriate

    - Further turn logic is easy to implement with use of chain of responsibility pattern
    - Dragon card and map piece features are harder to extend due to current implementation
    - Maybe need a third controller class for the settings page ?? not sure

Understandability of the solution direction [Appropriateness recognizability

- Very understandable
    - Design is easy to understand
    - Chain of responsibility and singleton design pattern requires a little further reading beyond just looking at the system if the user is unfamiliar

Extensibility of the solution direction (in anticipation of extensions to the game for Sprint 4) [Modifiability];

- Somewhat extensible, turns are extremely extensible, cards and mappieces will require more modification to the code compared to if implemented using inheritance

Quality of the written source code (e.g., coding standards, reliance on case analysis and/or down-casts) [Maintainability];

- Appropriate class and function names
- Appropriate documentation for complex functions

Aesthetics of the user interface [User engagement];

- Somewhat likely
    - Game board closely replicates real world design of the game
    - Includes hand-drawn pictures of animals in the game
    - Text alignment in settings page can be improved
    - Colour scheme is very bright - white, yellow, orange

Usability [Operability, self descriptiveness, user assistance]:

- Mostly understandable
    - Buttons have clear affordances
    - Slider follows natural mapping, 2 to 4 players from left to right
    - Clicking on a chit card flips it as expected
    - However user is never told chit card can be flipped by clicking, assumes user is already familiar with the game

Summary of key findings:

- Approach of displaying images for each tile and dragon card is ideal
- Turn logic using chain of responsibility design pattern is very applicable
- Current implementation of dragon cards and map pieces need to be modified for feature extensions

What are the ideas/elements of each of the tech-based prototypes from Sprint 2 we are going to use for Sprint 3?

- Chain of responsibility from Andrew's (Turn and turnLogic classes)
- Singleton Board class from Jeff's
- (Modified) MapPiece and Tile, Deck implementation from Krishna's
- All of our game/controller/main classes are similar

What new ideas/elements do we need to bring in in order to improve the prototype?

- Game instructions/user guidance/distinct ways to identify what is happening on the board
- Add an end turn button for situations where players want to end turn early

2. Object-Oriented Design

Board

The Board class in this design serves to closely mirror how the Fiery Dragons game would function in real life. Its role in this system is to be the 'container' for all relevant elements that would need to be interacted with, meaning it contains all relevant Dragon Cards by containing Deck and MapPieces.

The Board Class is implemented using the Singleton Design Pattern, meaning that only one instance of Board can be created, and ensures that all other classes that need to reference Board are able to, i.e. BoardController, Game, etc. This also means that the Board object is only initialised when it is first referenced.

Giving the Board class only these responsibilities means that we can avoid a god class, as currently it already stores all the most important components of the gameboard. By diverting all 'logic handling' to its collaborators, we can ensure that we maintain Single Responsibility Principle.

Responsibilities:	**Collaborators**
● Creating initial map pieces ● Creating initial deck	Game, MapPiece, Deck, Turn, BoardController

MapPiece

This class is used to represent the 8 map pieces that when combined together, make up the board. Each MapPiece will be composed of 3 Tiles. It has been created for the sake of abstraction, and to improve the extensibility of the Fiery Dragons game, keeping in mind the additions in the final sprint.

Each MapPiece will be responsible for initialising 3 tiles. Combined together, the 8 MapPieces composed of 3 Tiles each will make up the 24 Tile Board where the game is played. While the option of initialising this in the Board class was considered, it was rejected to uphold the Single Responsibility Principle. Additionally, having a MapPiece class would allow for future extensions specific to MapPieces, such as changing the number of Tiles per MapPiece.

Responsibilities: ● Initialising Tiles	**Collaborators** Board, Tile

Player This class is solely responsible for the action of flipping a chit card. When a card is flipped, this class will be responsible for telling the board what was flipped and thus letting the Turn class know how to proceed with the turns. An alternative was to have the BoardController class take responsibility of flipping the card, however the controller class's main purpose is to render, thus the player class was handed the responsibility of flipping chit cards.	
Responsibilities: ● Flipping the chit cards	**Collaborators** Turn

Game This class is responsible for handling all required game initialisation and allowing a player to flip a chit card. Once a chit card is flipped, this information is then passed to the Turn class for turn logic handling. Previously, the Game class would also handle the logic for determining whether a player had won, i.e., checkForWin(), however this responsibility was eventually given to turn logic handling, where the Chain of Responsibility pattern is used.	
Responsibilities: ● Initialising initial board state given input settings, e.g., number of players ● Stores Players array to keep track of players ● Handles player flipping a chit card ● Calls turn class to handle turn logic	**Collaborators** Turn, Board, Player, StartApplication, SceneController

Turn The main responsibility of this class is to play out a turn and handle any changing of turns from one player to another. For every chit card a player moves, the Turn class will figure out who is flipping the card from the Game class, and also get information of the Board class, which allows turns to be calculated and carried out. It will then end the turn and proceed to the next one. It was considered that the Turn class also handle all of the turn logic such as checking if a player is able to move from one tile to another. This was then distributed to other specialist classes to comply with SRP and not overcomplicate classes. The handling of next turn may also be completed by the Game class, but the Game class already had distinct responsibilities to perform.	
Responsibilities: ● Play out the turn ● Change of player from one to another	**Collaborators** Board, Game, TurnHandler

Deck This class is used to represent the 16 chit cards that are displayed in the centre of the Volcano. It's only responsibility is to initialise the chit cards with their default values. An alternative location for the initialising chit cards responsibility was the Board class, however to uphold SRP and for abstraction, the chit cards should be initialised here. This design choice additionally makes the game more extensible, should there be changes to types of Decks, or card allocations in Decks.	
Responsibilities: ● Initialising chit cards	**Collaborators** Board, TileType

BOARD - Jeffrey
MAPPIECE - Krishna
PLAYER - Andrew
GAME - Jeffrey
Turn - Andrew
DECK - Krishna

1. UML - Jeffrey and Krishna add respective parts Thursday 09/05
2. Sequence -
    a. Board init - Jeffrey
    b. Card flip - dont change
    c. Player moving depending on card - Andrew
    d. Turn to next player - Andrew
    e. Winning game - Krishna
3. CRC cards
4. Coding
5. Video demo
6. Git related stuff
7. Wiki
8. Commit analytics or some shit

MEETING 10/05 3pm - UML, Sequence DONE

1. setting up the board - **Krishna**
2. setting up the Game class, initialising playerList - **Jeffrey**
3. handling turns, flipCard(), Turn class - **Andrew**
4. End turn button - **Andrew**
5. **TurnHandlers - Andrew**
6. Changing to next player - **Jeffrey**
7. **rendering the movement of the dragon token on the board - Jeffrey**
8. some sort of message displaying whos turn it is - **Krishna**
9. **win game; rendering win game, (*maybe* button to restart game/new game) - Krishna**

To do: clean up all the classes in the code since a lot has changed