

# Fiery Dragons

## Sprint Four

Version 1.0

**Prepared by:**

Brian Nge Jing Hong

Wee Jun Lin

Ahmad Hafiz Bin Zaini

Sineth Fernando



## Contents

<b>1. Object-Oriented Design</b>	<b>3</b>
1.1 Revised Design	3
<b>2. Tech-based Software Prototype</b>	<b>7</b>
2.1 Build Instructions	7
<b>3. Reflection on Sprint 3 Design</b>	<b>8</b>
3.1 Extensions Incorporated	8
3.2 Difficulty to Incorporate Extensions	9

# 1. Object-Oriented Design

## 1.1 Revised Design

In Sprint 3, the team has designed the software architecture for the fully functional and playable implementation of the Fiery Dragon's game. Additional requirements have been presented in this Sprint (Sprint 4) which therefore necessitates the need to modify the class diagram. The additional requirements are:

1. Implement a new dragon card which can move the player back to the previous cave (see Section 3.1 for further information).
2. Implement game state management functionality that saves and loads information about the game (see Section 3.1 for further information).
3. Implement a timer functionality that imposes a 5-second limit on players to choose a dragon card (see Section 3.1 for further information).
4. Implement a tutorial walkthrough to guide players to learn the gameplay (see Section 3.1 for further information).

In this section, we present an updated class diagram that reflects all the extensions and modifications implemented since Sprint 3. This revised design addresses the necessary enhancements and any defects identified in the previous sprint, ensuring a more robust and comprehensive system architecture.

To facilitate understanding, changes from the Sprint 3 diagram are highlighted in red. These changes encompass new classes, modified attributes and methods, as well as adjusted relationships and cardinalities.

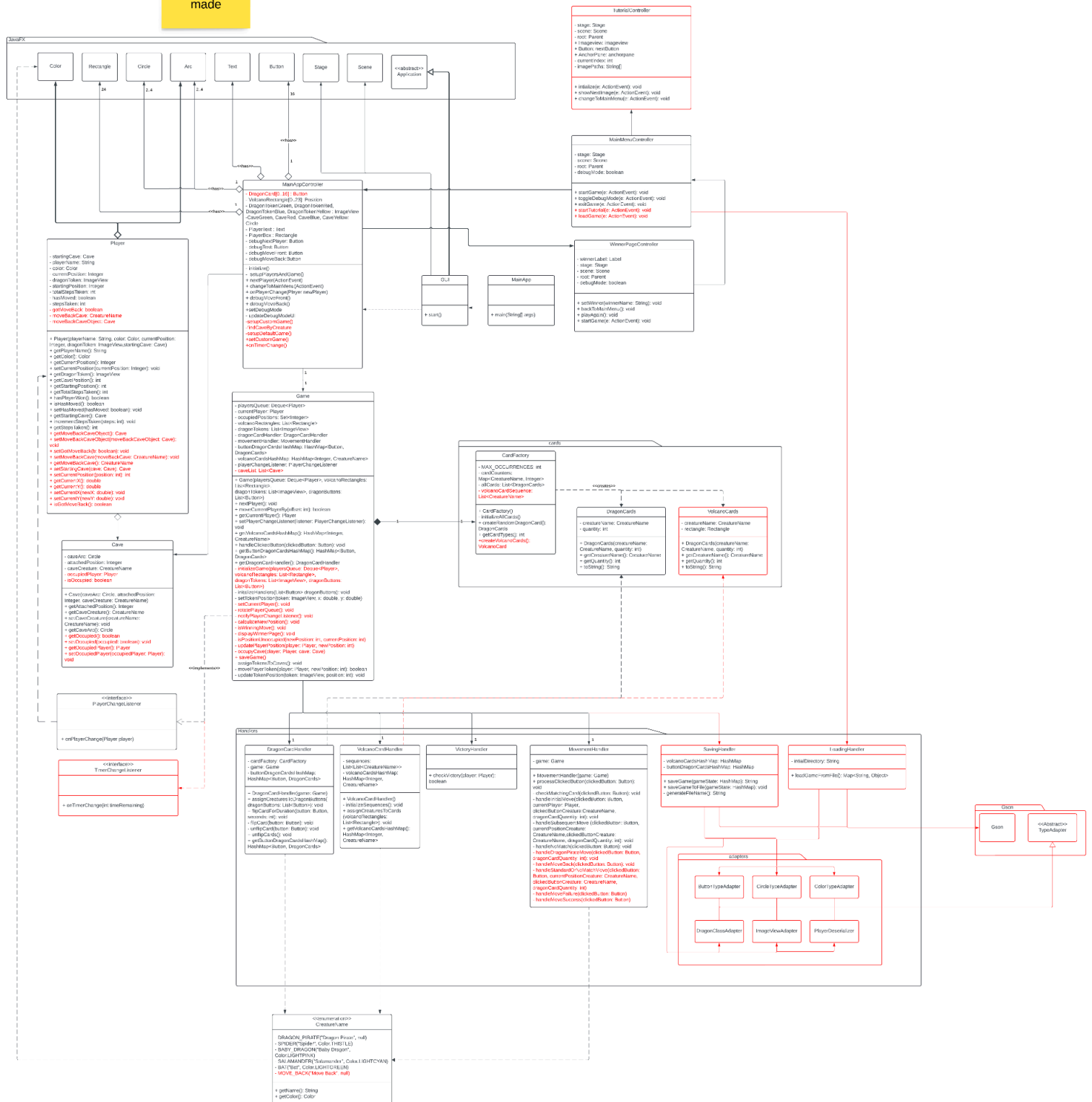
Reflecting on interaction patterns before coding is crucial for maintaining a well-structured and maintainable codebase. Therefore, while this document focuses on the updated class diagram, considering the sequence of interactions and data flow between objects has been a guiding principle throughout the design process.

The following pages present the updated class diagram, with clear indications of what has changed from the previous version. This comprehensive overview will serve as a foundational reference for the implementation phase, ensuring that all team members are aligned on the system's architecture and interactions.



# Dragonix

RED  
indicates the  
changes  
made



Lucidchart Link:

[https://lucid.app/lucidchart/a9b8f420-f1f4-455d-8e87-27855af1f369/edit?viewport\\_loc=-1795%2C-4487%2C3282%2C1720%2Ch\\_Cy4\\_7XBG-K&invitationId=inv\\_1dbd4ba3-87b7-4078-92b0-ada97597c17b](https://lucid.app/lucidchart/a9b8f420-f1f4-455d-8e87-27855af1f369/edit?viewport_loc=-1795%2C-4487%2C3282%2C1720%2Ch_Cy4_7XBG-K&invitationId=inv_1dbd4ba3-87b7-4078-92b0-ada97597c17b)

**Key Changes from Sprint 3:**

- **New Classes** (Added classes to support new features and enhancements)
  - TutorialController
    - Controller class for the tutorial page.
    - Private String[] imagePath, stores all the images of the tutorial explanation.
    - showNextImage, functionality for the next button to show the next image in the string of images stored.
    - changeToMainMenu, functionality to return back to the main menu.
- **Updated Attributes** (Adjusted attributes to fix previous defects and accommodate new requirements)
  - MainAppController
    - 15 to 16 Buttons in the DragonCard array.
    - startTutorial, this loads the tutorial scene.
    - timerText to display text of countdown.
  - Player
    - gotMoveBack to indicate if the player has been moved back to the cave.
    - moveBackCave to indicate the creatureName that the player has to get in order to move out of the cave.
    - moveBackCaveObject to reference the cave object that the player has been moved back to.
  - Cave
    - occupiedPlayer to indicate the player occupying the cave.
    - isOccupied to indicate if the cave has been occupied or not by players.
  - Game
    - caveList to find the nearest cave to the player in cave order.
    - timerChangeListener to set timer change.
    - Timeline to record the start and end so that we can find the timeRemaining.
    - timeRemaining to set time remaining.
  - CreatureName
    - MOVE\_BACK to indicate if the dragon card player chooses is a move back card.
- **Modified Methods** (Revised method signatures and implementations to enhance functionality)
  - Player
    - getMoveBackCaveObject to get the reference to the cave object that the player got moved back to.
    - setMoveBackCaveObject to set the player's move back cave to the one that he/she is sent to.
    - setGotMoveBack to indicate that the player has been moved back to the cave.
    - getMoveBackCave to get the CreatureName of the cave that the player has been sent back to.
    - setStartingCave to set the starting cave of the player.

- setCurrentPosition to set the current position of the player.
- getCurrentX to get the current x position of the player.
- getCurrentY to get the current y position of the player.
- setCurrentX to set the current x position of the player.
- setCurrentY to set the current y position of the player.
- isGotMoveBack to determine if the player has been moved back to a cave.
- Cave
  - getOccupied to determine if the cave has been occupied.
  - setOccupied to set the cave to becoming occupied.
  - getOccupiedPlayer to get the player that is occupying the cave.
  - setOccupiedPlayer to reference the player to occupy the cave.
- Game
  - initializeGame to decouple and reduce the cyclomatic complexity of the constructor method that was used in Sprint 3.
  - setCurrentPlayer to set the current player at the start of the game.
  - rotatePlayerQueue to rotate between the players in the queue.
  - notifyPlayerChangeListener to notify if there's changes in player's turn.
  - calculateNewPosition to find the new position of the player's token on the volcano cards.
  - isWinningMove to determine if the current player has won.
  - displayWinnerPage to redirect to the winner page.
  - isPositionUnoccupied to determine if the volcano card is unoccupied
  - updatePlayerPosition to move the player's token to their respective positions.
  - occupyCave to allow the cave to be occupied by the current player.
  - startTurn to start 5 seconds timer for each player's turn.
  - initializeGameTimer Initialises the game timer for the first player's turn.
  - setTimerChangeListener to set timer change.
- MovementHandler
  - handleDragonPirateMove to handle the movement if the dragon pirate card was chosen by the player.
  - handleMoveBack to handle the movement if the move back card was chosen by the player.
  - handleStandardOrNoMatchMove to handle the movement if normal cards such as spider, baby dragon, salamander or bat were chosen or not chosen by the player.
  - handleMoveFailure to hide the dragon card.
  - handleMoveSuccess to make sure the same dragon card cannot be chosen again.
- MainAppController
  - onTimerChange to set the text on the timerText display.

## 2. Tech-based Software Prototype

### 2.1 Build Instructions

This section provides comprehensive instructions for building the Fiery Dragon game executable, focusing on the Windows platform but applicable to other platforms as well. These guidelines are intended to assist developers and stakeholders in setting up and accessing the software prototype efficiently. For optimal performance and compatibility, ensure that you have "Oracle OpenJDK version 22" and JavaFX version 21 installed on your system. Alternatively, you can use Liberica JDK, which includes JavaFX by default. These prerequisites are essential to ensure the successful execution of the program.

We provide two methods for building and running the Fiery Dragon game executable: using Maven and using IntelliJ IDEA. Each method is detailed with step-by-step instructions to guide you through the process.


#### Maven Build Instructions

1. Download Maven and ensure that the installation is successful with `mvn -version`.
2. Navigate to the Project Directory
3. Open the terminal and run the following commands:
  - a. `mvn clean compile`
  - b. `mvn javafx:run`
  - c. `mvn javafx:jlink`

#### IntelliJ Build Instructions

1. Open the Project Structure dialog by navigating to File>Project Structure.
2. Navigate to the Artefacts tab and add JAR, selecting "From modules with dependencies" from the dropdown options.
3. Select the Main Class with the public static void main method. Make sure that the option "extract to target JAR" is selected.
4. Click on "OK" button
5. Navigate to "Build" on the toolbar and click "Build Artefacts".
6. The executable file will appear inside the "out" folder in the project directory..

To further assist with the setup, we have included a video tutorial demonstrating both methods—using Maven and using IntelliJ IDEA. This video offers a comprehensive guide to ensure a seamless build process.

 2024-05-14 22-54-40.mkv

By following these instructions, users can effortlessly build and run the Fiery Dragon game executable on the Windows platform, leveraging Oracle OpenJDK version 22 and JavaFX version 21 for a seamless experience.

In this sprint, the additional extension of saving the game state has been implemented. Users can click on the "save" button during the game. After saving the game state, the saved game state would appear in the "saves" folder in the same directory as the executable file. The saved game state can then be loaded from the "saves" folder by selecting the desired game state when prompted.

## 3. Reflection on Sprint 3 Design

### 3.1 Extensions Incorporated

In this section, we detail the specific extensions incorporated into the system during Sprint 4. Each extension is discussed in terms of its purpose, the changes made to the system, and the challenges encountered during implementation. This analysis provides insight into how the initial design from Sprint 3 influenced the integration process and highlights the improvements made to enhance the system's functionality and robustness.

#### **Extension 1 (New Dragon Card)**

When this card is flipped, the player's token must move backward until it finds an empty cave, positioning itself in this cave. The player does not lose their turn and can continue flipping the next Dragon card. If the token is already in a cave, it remains there, and the player's turn continues.

#### **Extension 2 (Game State Management)**

We implemented the functionality to save and load the game state from an external text file in the .json format. This file will store the following:

- All Volcano cards (including animals on tiles and their sequence)
- Sequence of Volcano cards forming the circular Volcano
- Location of caves and the animals in them
- Location of each player's token
- Location of each Dragon card
- Order of players and the current player's turn

The format chosen will support different numbers of animal tiles on Volcano cards, different locations for caves, and a variable number of Volcano cards. The format must also support any future extensions.

#### **Extension 3 (Timer Functionality)**

We introduce a timer functionality to add a layer of urgency and excitement to the game. When it is the player's turn, the timer functionality imposes a 5-second limit on players to choose a dragon card. The player would lose their turn if the player doesn't choose a dragon card within the 5 seconds. If the player selects the correct dragon card and is able to continue their turn, the 5 seconds timer resets and the player can pick another dragon card.

#### **Extension 4 (Tutorial Walkthrough)**

For the fourth extension, a tutorial walkthrough would be created and made available for players to learn the gameplay of "Fiery Dragons". This interactive guide would help new players learn the game's mechanics and rules, making it easier for them to get started and enjoy the game. The interactive guide would have buttons that players can click on that will progress a slideshow of screenshots of the game that showcases different game environments, so that players can understand the variety of options the game produces and what to do in those situations. The option to enter the tutorial walkthrough mode would be made available in the game menu page.



## 3.2 Difficulty to Incorporate Extensions

In this section, we reflect critically on the design and implementation decisions made during Sprint 3, focusing on how these decisions influenced the ease or difficulty of incorporating new extensions in Sprint 4. This analysis is crucial for identifying areas for improvement and ensuring more flexible and scalable design practices in future development cycles.

### **Extension 1 (New Dragon Card)**

Adding the extension of having the new dragon card presented a medium level of difficulty. The primary challenge was understanding the existing code related to movement, as changes were made to the movement class in Sprint 3, and not all team members were familiar with these modifications. Therefore, it took some time to understand certain classes which made modifications and extensions more complex.

The separation of core functionalities into different classes in Sprint 3 definitely facilitated easier extensions in Sprint 4 by adhering to the Single Responsibility Principle (SRP). However, the initial learning curve was steep due to uneven knowledge distribution among team members. If we were to revisit Sprint 3, a crucial improvement would be to ensure comprehensive knowledge transfer among team members. This could be achieved through a dedicated meeting to explain the rationale behind each class, ensuring everyone understands the design and where to extend their chosen features. This strategy would prevent unnecessary refactoring and promote a shared understanding of the codebase, enhancing the team's ability to implement extensions efficiently. It also ensures that everyone understands the distinct and clear purpose that each class already has, preventing any unnecessary refactoring.

### **Extension 2 (Game State Management)**

Saving and loading the game presented a very high level of difficulty. The primary challenge was the Game using hashmaps to store the game data with JavaFX objects serving as the keys to the game values. This proved challenging when trying to save the hashmaps into our chosen format of .json using the Gson library. This is because Gson struggles to serialise Objects correctly into a .json format similarly it struggles to deserialize .json data back into Objects without using special classes called adapters. We overcome this obstacle by creating new Objects based on the loaded data instead of loading the data directly from the save file. Another issue was that in Sprint 3, we hardcoded the game board with a specific number of Volcano and Dragon Cards using SceneBuilder in JavaFX. Therefore, we struggled to have a dynamic number of VolcanoCards and Dragons, however in theory our Game should be able to handle this.

If we could start over Sprint 3, we would design the game board dynamically within the MainAppController rather than injecting a pre-defined FXML file. An example of this would be to create new Rectangle and Button objects in the MainAppController and add it into scene instead of injecting the FXML file into the MainAppController. This approach would allow for a more flexible and scalable design, accommodating dynamic changes in the game state without significant rework. Moreover, utilising more suitable data structures and serialisation methods from the beginning would simplify future extensions and maintenance.

### **Extension 3 (Timer Functionality)**

The extension of implementing a timer functionality presented a medium level of difficulty. The primary challenge encountered was grasping the `Timeline` class in JavaFX, especially in conjunction with `KeyFrame` and `Duration`. Understanding how to create and manipulate these objects to achieve desired animations requires a moderate learning curve. The complexity lies in comprehending the sequence and timing of events within the `Timeline`, and how `KeyFrame` actions are scheduled and executed based on `Duration` intervals. Additionally, integrating this with JavaFX's FXML to dynamically display a countdown timer adds another layer of difficulty. Mapping functionality to the FXML involves setting up appropriate bindings and listeners to ensure real-time updates. This was particularly tricky until a listener mechanism learned from a previous sprint was utilised effectively. Since we used a listener mechanism in Sprint 3, we were able to use it as reference to implement the timer functionality, which made this extension easier to implement comparatively. This listener facilitated the linking of backend logic with the FXML interface, enabling the countdown timer to reflect accurately, thus overcoming a significant hurdle in the implementation process. If we could revisit Sprint 3, we would consider adding a clock component in the Game class. This addition would lay a foundation for timer-related functionalities, making their implementation in Sprint 4 more straightforward. However, we would still be continuing to use listener mechanisms for linking backend logic with the FXML interface as it proved effective and would remain a best practice for future extensions.

#### **Extension 4 (Tutorial Walkthrough)**

The extension of implementing a tutorial walkthrough presented a fair level of difficulty. This extension of adding the tutorial was relatively straightforward due to the modular design and adherence to SRP in Sprint 3. We just needed to follow the implementation of the winning page as the tutorial pages followed the same structure. The tutorial implementation only required the addition of one extra class and the modification of one existing class, the MainMenuController class. This showcases the flexibility of the codebase as well as the adherence to SRP. Therefore, it is evident that following best practices like modularity and SRP can significantly ease the process of adding new features. Sprint 3's planning for scalability and flexibility as well as improvement in the codebase design prevented extensive refactoring for this sprint, Sprint 4 and especially in the implementation of the tutorial walkthrough. Through the challenges learnt in Sprint 3 of extending the code, towards the end of Sprint 3 we were able to create a structured codebase that adhered to practices like modularity and SRP to improve flexibility in code, allowing us in Sprint 4 to extend our code without the complications of affecting already existing code as well as with ease. If we could go back to Sprint 3, we would enhance the modularity and scalability of the MainMenuController class further. By refining the methods and improving the SRP adherence, we would ensure even easier extension capabilities for future features. The structured codebase from Sprint 3 significantly eased the process in Sprint 4, but continuous refinement in line with SRP principles would yield long-term benefits.

#### **Conclusion**

Reflecting on our design and implementation choices in Sprint 3 reveals the importance of shared understanding, flexible design, and adherence to best practices like SRP and modularity. Ensuring comprehensive knowledge transfer among team members, designing for scalability, and refining code modularity are key strategies for improving future development cycles. These reflections highlight areas for improvement and guide us towards more effective and efficient extension implementations in subsequent sprints.