FIT3077: Software Engineering: Architecture and design

Semester 1, 2024

Reflection

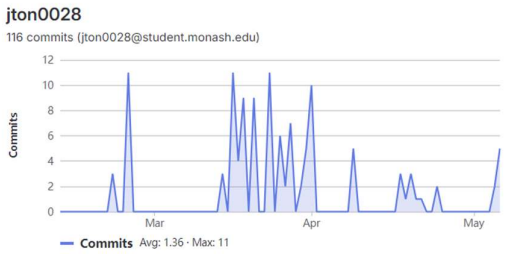
Group No: MA_Wednesday_04pm_Team690

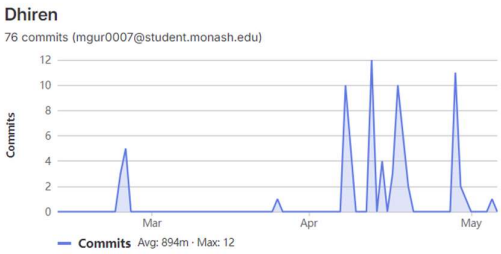
Prepared by:

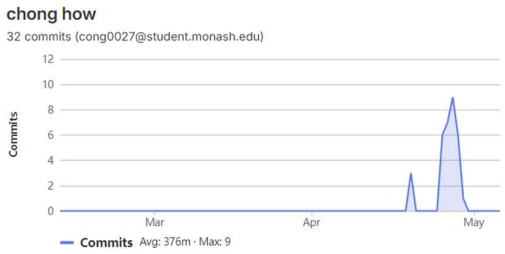
Tong Jet Kit	32632177	jton0028@student.monash.edu
Mandhiren Singh Gurdev Singh	32229828	mgur0007@student.monash.edu
Nisha Kannapper	31121993	nkan0018@student.monash.edu
Ong Chong How	33363102	cong0027@student.monash.edu

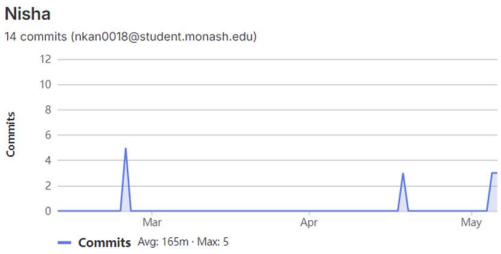
Content

Contributor Analysis









Extensions Done

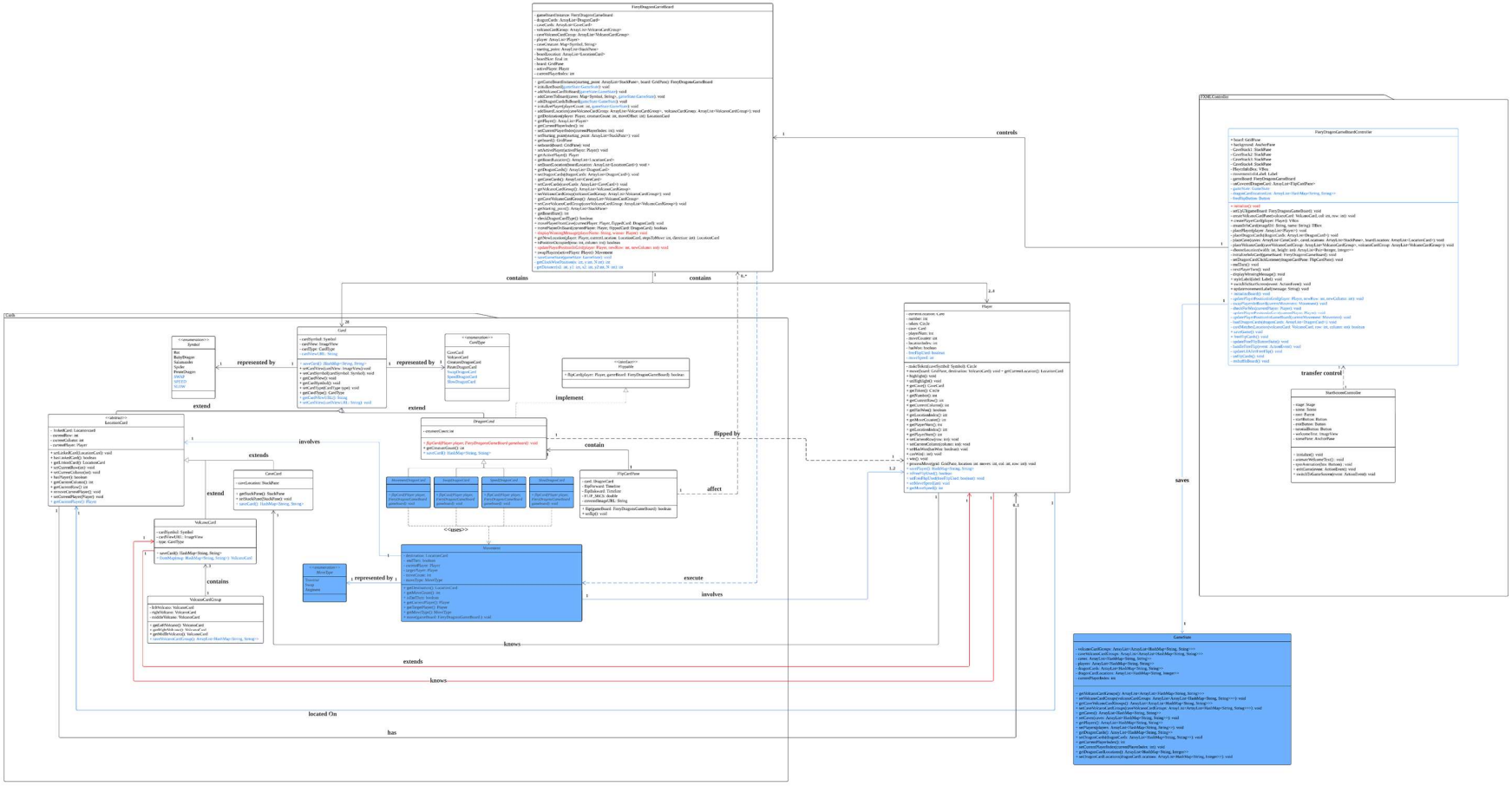
In Sprint 4, we have incorporated four extensions into our project. These include the two required extensions specified in the assignment and two additional self-defined extensions, one of which addresses a Human Value. Below, I will outline the self-defined extensions we have created:

a) Reveal Card Power Button
   - **Description**: A button that allows each player to reveal all cards on the board momentarily, providing them with an opportunity to memorise the locations of the cards on the board. However, each player can only use this powerup once per game.
   - **Purpose**: To allow users to be able to plan ahead and use the button only when absolutely necessary.
   - **Human Value Addressed**: Fairness, Pleasure and Engagement. By allowing players to view all the dragon cards momentarily, it has provided an opportunity to make the game more fun as it increases player engagement by providing a strategic action they could use in game. Moreover, since each player can use this button once, they will all still have an equal opportunity to win the game.

b) Speed Augment Dragon Cards
   - **Description**: Two additional dragon cards were added to modify "player move speed". Flipping the Speed card will use a multiplier on all steps taken forwards and backwards, while flipping the Slow card halves all steps taken forward and doubles all steps taken backwards. After one of these two cards are flipped, all dragon cards are reshuffled.
   - **Purpose**: To add variation and an element of difficulty to gameplay.
   - **Human Value Addressed**: Ambitious, Daring, Intelligent, and Variation in Life. By allowing a reshuffling of the deck, more variation in gameplay can occur, since flipping one of the two cards reshuffles all dragon cards, so anything players have memorised is no longer applicable. Additionally, flipping a Speed Augment card affects a player's next movement, so a player will move a significant distance forwards or backwards, which can be a great benefit or irritant. When paired with the Reveal Cards button, players can carry out ambitious strategies to win.

UML Diagram

Lucid Chart Link: https://lucid.app/lucidchart/5f53778e-44a8-43c5-bacf-0101f3127e71/edit?viewport_loc=-188%2C-1565%2C5657%2C2616%2Cg2vE-5gN-W9k&invitationId=inv_4e782b54-80fa-4066-83c8-077067e3a230



Reflection

Sprint 3 was pivotal in establishing a robust class hierarchy, encapsulating game elements such as volcano cards, cave cards, players, and dragon cards within dedicated classes. Each class was designed to encapsulate properties and behaviours specific to its corresponding game element, following the principles of encapsulation and modular design. In Sprint 4, we have to extend our tech game prototype that we have created in Sprint 3. Our team is a team of 4 so we have created 4 different extensions which are a new dragon card, load and save function and 2 creative extensions where one needs to contain human values. There were some challenges while implementing the extensions but there were some parts where it was easy as the Sprint 3 prototype accounted for flexibility.

Incorporating the functionality of the newly implemented dragon cards was not as difficult as the other extensions to integrate with the Sprint 3 prototype. This is due to the UI and base game structure used in Sprint 3 can accommodate the addition of new dragon cards. The classes used for the card flipping functionality are also structured well for flexibility. For example, there is an interface class called “Flippable " which allows any component of the game that can be flipped to resolve its flip action by implementing this interface in its class. Moreover, there is also the FlipCardPane class which facilitates the integration of any dragon cards into the UI with a flipping animation to visually indicate when a flip has occurred. With this class and interface, it was easy to add the new dragon cards into the UI and the Model as Sprint 3 has accommodated for any future components that can be flipped including a new dragon card. Additionally, rigorous testing and continuous feedback during the implementation phase in Sprint 2 by Tong Jet Kit has helped identify and resolve any potential issues early.

The challenging part of implementing the “Swap” and “Speed Augment” cards was that the new dragon cards required an overhaul of movement handling and card flip handling. This was needed due to our current implementation being designed for the basic dragon card, which only handles player movement. Therefore, a new Movement class and MoveType enum was designed and integrated into our Sprint 4 codebase.

The first new dragon card, the required extension “Swap” dragon card, needed a way to resolve the swap action after flipping the card. Our Sprint 3 prototype, designed for the basic dragon card, only handled movements where the player moves forward or backward based on the card that is flipped. Additionally, since our win condition is based on the total number of steps a player has moved on the board, the swap dragon card necessitates updating the step counts for both the active player and the swapped player. This is challenging because the active player might

be in a cave, and updating the move counts for the players varies depending on their positions on the board.

The second and third new dragon cards, the self-extension "Speed Augment" cards, required the main game controller class to know what type of card had been flipped, in order to handle the unflipping and reshuffling of cards. This necessitated a change in the way the card classes themselves handled a card flip, since we needed to wrap the data about the card flip in a meaningful way to be returned.

For the load and save required extension, incorporating it into the Sprint 3 design and implementation was moderately challenging due to the lack of initial planning for persistence. While the modular design and encapsulation principles we followed in Sprint 3 provided a good foundation, the absence of dedicated serialisation mechanisms made it difficult to implement the load and save features smoothly. We had to retrofit serialisation logic into existing classes, which introduced some complexity and potential for bugs. One of the major difficulties stemmed from ensuring all game state data, including player positions, card states, and game progress, could be accurately captured and restored. This necessitated meticulous attention to detail and rigorous testing to safeguard against any loss or corruption of game state data during the save and load process. However, amidst these challenges, certain aspects did facilitate the integration of load and save functionality. For instance, the modular design allowed us to isolate the components related to game state data, making it easier to identify and implement serialisation logic for these specific elements. Additionally, the encapsulation principles ensured that the state data was encapsulated within appropriate classes, reducing the risk of unintended modifications or data corruption. Overall, while challenges existed, leveraging our modular design and encapsulation principles helped mitigate complexities to some extent.

For the other human value extension, we decided to add a "Reveal Cards" power in the form of a button on the user interface (UI). This button allows each player to reveal all cards on the board momentarily, providing them with an opportunity to memorise the locations of the cards on the board. However, each player can only use this powerup once per game. Therefore, they would have to plan ahead and use the button only when absolutely necessary. In terms of the human values, this ensures "Fairness", in which each player has an equal opportunity to use the free flip. Besides that, there are also the human values of "Pleasure" and "Engagement", where each player has the opportunity to get excited to use this powerup in order to have a more fun time, as well as it increases player engagement by providing a strategic action they could use in the game. The human value of "Communication" is also seen, as the UI clearly communicates the game state changes to the player.

Incorporating this functionality into the design and implementation from Sprint 3 presented a few challenges, however overall it was not that hard to incorporate it. The easy aspects when developing this functionality are in terms of the GridPane structure we used in Sprint 3, as well as the idea of "separations of concerns" too. For the GridPane structure, we were able to use the existing GridPane for the game board, which made it quite straightforward to iterate over the cards in the GridPane to apply the flipping functionality. This structure allowed for easy access to each node in "FlipCardPane", which helped immensely with the flipping process, as we could just reuse the code there to flip and unflip the cards when the powerup is used. This let us reduce redundant duplicated code and we were able to just reuse the code from Sprint 3, mostly. Besides that, in terms of the separation of concerns, the separation of game logic and UI components in different classes helped us maintain the clarity of the code when implementing the "reveal cards" functionality. This is because the "FieryDragonsGameBoard" class handled the game state, while the "FieryDragonsGameBoardController" managed user interactions, which included the "reveal cards" human value functionality.

The difficult aspects when developing this functionality were the state management and interaction with existing game mechanics. In terms of the state management, we needed to ensure that each player could only use the free flip once, which undoubtedly required close attention to the state management. This process involved tracking the usage status for each player using the freeFlipUsed boolean attribute in the Player class, then updating the UI accordingly using the controller class for the game board. Besides that, in terms of the interaction with existing game mechanics, like card flipping and turn management, did add a little bit of complexity to the codebase. This is because we needed to ensure that the free flip did not disrupt the flow of the game or introduce bugs. This issue was caused by the multiple conditional branches when moving the player using the movePlayerOnBoard() and movePlayerFromCave() method, which calls the nextPlayerTurn() method in it and therefore it made it rather difficult to track if the player has used the free flip or not. One of the bugs that arose was that the "Reveal Cards" button could only be used once per game, by the first player who used the power up. This issue was caused by the convoluted code as mentioned earlier, and therefore we needed to have a couple of additional booleans to update the freeFlipUsed boolean in the Player class correctly.

If we were to go back to Sprint 3 to redo our implementation so that the extension for Sprint 4 would be easier, what we would do differently is put the responsibility for updating the user interface of the gameboard to the controller class instead of the gameboard class. This is because based on the Model View Controller architecture, the controller class should be in charge of resolving any events by the player and then render the output to the View while the Model should just deal with data driven logic which in our case is determining if the player can move and update the player data after moving. Thus it was quite challenging to move the responsibility from

one class to another without causing additional bugs to surface. Alternatively, we could have designed a class specifically to characterise player movement. This approach would be particularly beneficial for the new dragon card that swaps players' positions. Our current implementation is tailored for the basic dragon card, which only handles movements where the player moves forward or backward based on the flipped card. Hence it is quite inflexible if there are other types of movement in future sprints. This issue can be seen from this Sprint 4 where there is a new dragon card. This new dragon card introduces a different type of movement by swapping players' positions, so having a dedicated movement class would streamline this process. This class would encapsulate all movement logic, making it easier to accommodate various types of movements, including swaps which the new dragon card will do.

Furthermore, in terms of the reflection on Sprint 3's design and implementation, we can see that the distribution of system intelligence, with most of the game logic centralised in the "FieryDragonsGameBoard" class, allowed for easy addition of new game rules like revealing all the cards. However, this centralization did sometimes make it challenging to manage the interactions between the different components of the game. Besides that, there was a presence of code smells, where certain code smells like the long movePlayerFromCave() and movePlayerOnBoard() methods made the extension more difficult to implement as there were multiple conditional branches in these methods that caused tight coupling of certain game components like moving the player and changing to the next player's turn. Another example is that managing the game state and updating the UI in different parts of the code led to a few inconsistencies and bugs as mentioned in the previous paragraph. In terms of the use of an appropriate design pattern in Sprint 3, the existing design pattern of Model-View-Controller (MVC), actually helped facilitate the addition of this extension immensely. This is because the separation of concerns that is inherent in the MVC pattern allowed for the reveal cards functionality to be added to the controller without modifying the game logic significantly.

In terms of improvements for future sprints to facilitate new extensions like revealing all cards, we believe that we can simplify certain methods like breaking down long methods into smaller, more manageable pieces. For example, we could further modularise the movePlayerOnBoard() and movePlayerFromCave() methods to isolate movements like backward and forward movement using an interface instead of the multiple conditional branches which made it harder to debug the "reveal cards" functionality. This would involve refactoring methods in the controller and game board classes to adhere to the Single Responsibility Principle (SRP). Besides that, we could also reduce the tight coupling between classes by introducing more interfaces and dependency injection would allow for easier integration of new features. For example, we can have an interface called MoveStrategy that has a move() method, then we make two classes, namely NormalMove and PirateMove that implements MoveStrategy, which will then

allow the different movement logic to be modularised and contained within these classes. Then, if we were to add more special DragonCards in the future, we would just have to implement the MoveStrategy interface. This would ease the debugging process for the reveal all cards extension as it would allow us to isolate issues with the code (whether it is a problem with the movement, forward or backward, affecting the revealing of cards) and work on debugging smaller parts of the code instead of the long methods with multiple conditional branches. This is called the “Strategy Pattern”, by defining a family of algorithms as mentioned above. This pattern would simplify the game logic and facilitate easy modifications to the rules. We could have also used the “Observer Pattern” to manage the interaction between the game state and the UI. This would ensure that changes in the game state automatically update the UI, which reduces the need for manual UI updates and decreases the likelihood of inconsistencies. Besides that, for future sprints, we could also have comprehensive documentation of the codebase. This would make it easier for future developers to understand and extend the system. This includes inline comments, design diagrams and user guides. Automated testing could also help. By implementing a suite of automated tests, including unit tests and integration tests, it would help us catch bugs early and ensure that new features do not break existing functionality. This would involve setting up a CI/CD pipeline to run tests automatically.

For load and save extension, several changes in Sprint 3 would have streamlined our implementation process. First and foremost, creating a dedicated GameState class to encapsulate all facets of the game state, such as player positions, card states, and current game progress, would have been invaluable. This centralization would have simplified the serialisation and deserialization processes necessary for saving and loading the game state. For example, the VolcanoCard class was designed to encapsulate its properties and behaviours, which makes it straightforward to extend with methods for serialisation and deserialization. Adding methods like saveCard() and loadCard() for file I/O operations is facilitated by this encapsulated design. Additionally, ensuring a clear separation between game logic and UI components would have facilitated serialisation without concern for the UI's state. Utilising interfaces and abstract classes to define game logic independently of the UI would have further optimised our approach. Implementing the Serializable interface (or an equivalent serialisation mechanism) for all relevant classes early on would have greatly eased the process of saving and loading the game state later. Assigning unique identifiers to each game object would have ensured accurate object references during these processes. Furthermore, abstracting file input and output operations through the design of interfaces or utility classes for reading and writing game state data would have streamlined the addition of save and load functionalities. Including versioning information in the saved game state would have enabled the system to gracefully handle future changes to the game state format, bolstering the game's robustness to updates and new features.

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

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