FIT3077 - Architecture and Design

Sprint Four

MA_Tuesday08am_Team123

Swift Crafters

Team Members:

Maliha Tariq

Loo Li Shen

Khor Jia Wynn

Dizhen Liang

MONASH University

Table of Contents

1. Extensions Implemented 2

2. Object-Oriented Class Design 3

3. Reflection on Sprint 3 Design 4

3.1 Extension 1 - New Dragon Card 2 4

3.2 Extension 2 - Loading and saving to an external configuration file 6

3.3 Self-Defined Extension 3 - Equality Boost 8

3.4 Self-Defined Extension 4 - Shop with purchasable items 9

4. Retrospective on Sprint 3 Design/Implementation 10

5. Future Strategies and Techniques 11

6. Executable Instructions 13

7. Sprint Contributions 13

Wiki GitLab Link 13

Contributor Analytics 13

8. References 14

MONASH University

1. Extensions Implemented

Extension 1: Introduces a new Dragon Card that causes the player's token to swap positions with the nearest token not in a cave. The player then loses their turn. If all other tokens are in caves, no swap occurs. This adds strategic depth by considering token positions and the consequence of losing a turn.

Extension 2: Allows the game state to be saved to and loaded from an external configuration file. This includes the UI elements and the state of VolcanoCards, caves, dragon tokens, chit cards, player order, and the current player's turn.

Self-Defined Extension 3: Adds an "Equality Boost" card for players who reveal incorrect cards three times in a row. This boost lets them move one step forward automatically on their next turn and still take their turn normally, balancing the gameplay.

Self-Defined Extension 4: Introduces a shop where players can buy effects using points earned from making correct moves. This enhances gameplay by providing strategic options and encouraging players to plan their point distribution based on the game situation.

MONASH University

2. Object-Oriented Class Design

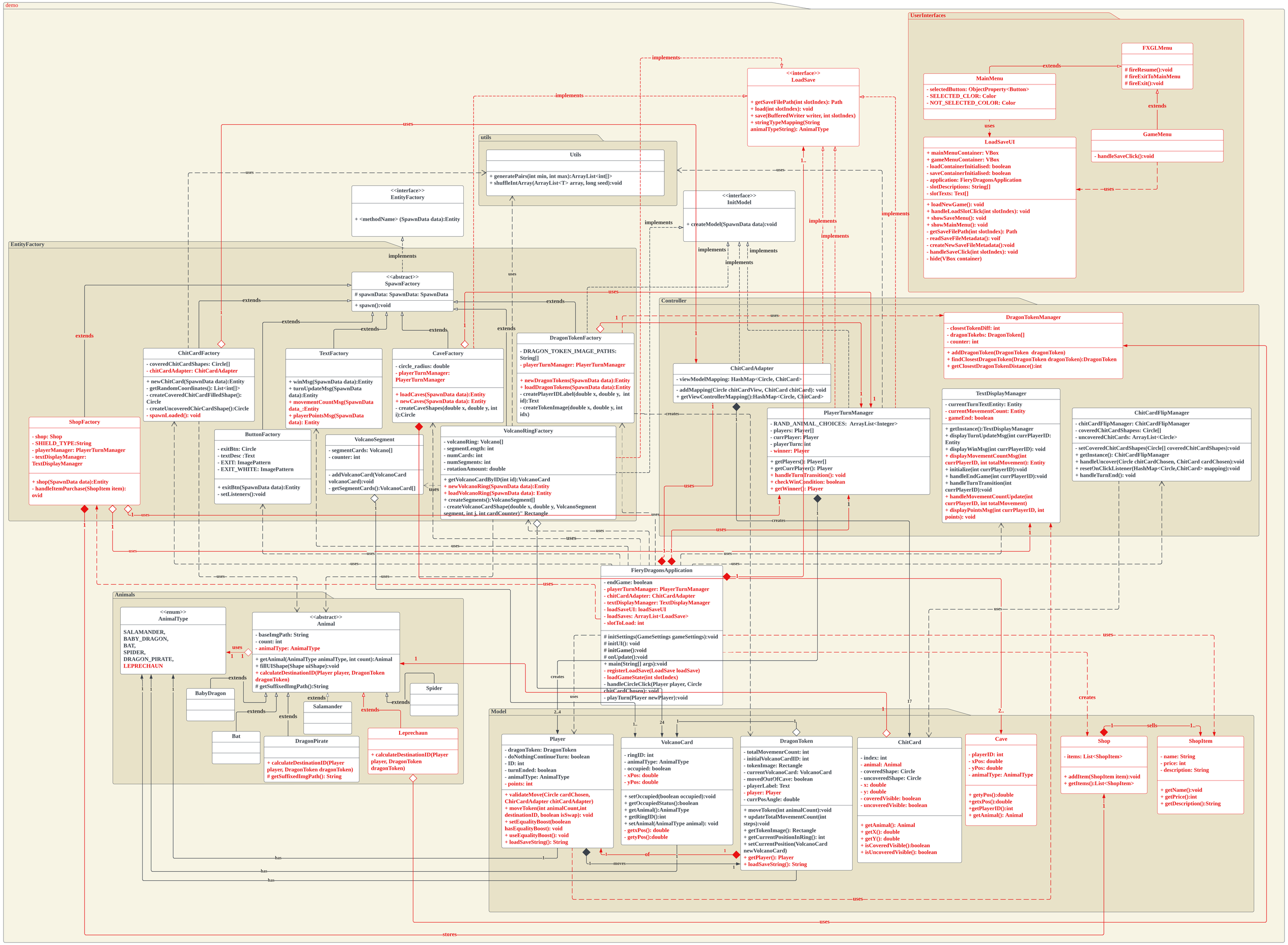
Notes:

1) The new sections on the class diagram are in **red**.
2) As per previous feedback:
    a) the composition/association relationship arrows now end with an arrow.
    b) 'Implements' relationship now uses a dotted arrow
    c) The chit card flip animation during the end of a player's turn is increased slightly
3) Generally, if a class inherits from a super class, or implements an interface, any inherited methods (or default methods of the interface) will not be shown. If it is shown, this means the class overrides the method with its implementation.
4) Previous feedback indicated that the AnimalType enum could violate the Open Closed Principle. However, in our latest design, whenever additional animal types are introduced, only three modifications need to be made. (In fact this is what we did with the introduction of the Leprechaun animal in this Sprint)
    a) Creating the class for this new animal and extending the Animal abstract class
    b) Adding the new enum to the AnimalType class (Like how we added AnimalType.LEPRECHAUN for this sprint)
    c) In the Animal abstract class, modify the getAnimal factory method to include this new case.

    We were taught to avoid the use of 'instanceOf' in FIT2099, and employ double-dispatch or multiple dispatch is not feasible.
    The key observation here: No other parts of the code need to be modified, and existing logic involving comparing animal types can simply check the enum (no extensive case analysis required)

5) Some differences from the Sprint 3 class diagram are not related to Sprint 4, but because they involve minor refactoring of methods for better modularity. These are also in red.

2. Class Diagram



MONASH University

3. Reflection on Sprint 3 Design

In this section, we reflect on Sprint 3's design and how it impacted our work on implementing extensions for Sprint 4. We first discuss the 4

3.1 Extension 1 - New Dragon Card 2

Description

The token of this player swaps position with the closest token to it, but this player loses their turn.
Constraints:

1) Any closest token that is in its cave cannot be swapped with. (This implies that if all the other tokens are in caves, no swap occurs). However, if a token is in its cave, and the closest token is not in a cave, then a swap can occur. [1]
2) If a token is close to its cave after having almost gone around the Volcano, it may have to go around the Volcano again.

Level of difficulty

Moderate. Overall, the design was very extensible. The only major design difference was introducing a DragonTokenManager that tracks the DragonTokens on the board in an array. Finding the closest dragon token would then involve this manager searching its array, computing the distance differences, and returning the closest token.

Factors contributing to difficulty

The difficulty lies mainly in implementing the logic for constraint (2) in the 'Description' section above. The algorithm used in the source code required careful analysis of the different scenarios in which swaps could be performed and looking for patterns. When indexes from 2 to 6 are involved, 'normalising' needs to be performed by temporarily adding 24 (the number of volcano cards in the ring) to these indexes to ensure correctness. Whenever a successful swap is to be performed, we check if that token's cave is in its path (i.e. in between the two swapped tokens), and we have to apply a lot of case analysis.

Reflection on what made this extension easy/difficult to address

What worked well to support the implementation of this extension:

- The initial complexity/redundancy of defining the Animal class abstraction has been paid off in this extension. Here, the Leprechaun class is easily integrated by extending this Animal class.
- The procedure for handling the results of a chit-card flip is extensible. The idea behind delegating responsibilities to different classes
    - Determining the destination VolcanoCard: Player
    - Moving the dragon token: DragonToken
    - Updating the state of the Model: VolcanoRingFactory, DragonToken

  proved to help with code maintainability, as it is significantly easier to handle smaller, modular methods than having to parse the logic of a bloated method.
- Choosing to define a dragon token class to encapsulate the information of where the dragon token is currently at, and whether it has moved out of a cave has paid off. This is because for the Leprechaun card, we need to access information about whether a dragon token is in its cave, and we need to know where it is currently.

Design Deficits:

- The original implementation used case analysis when validating the player's move. Specifically, the if-else statement in the Player class's makeMove() method handles the case separately for the DragonPirate. In hindsight, this cluttered the method, and this technical debt worsened with the introduction of the new Leprechaun-type dragon chit card with its unique implementation. This would require adding a third case for analysis.
- Choosing to use a variable to count the total movement count of a dragon token, and using it to check a win condition can still be applied even with constraint (2) above. However, the algorithm for updating the total movement count of dragon tokens after a swap is complicated, as seen in the source code.

MONASH University

3.2 Extension 2 - Loading and saving to an external configuration file

Description

The state of the game needs to be saved into a file, and it can be loaded from that file. Essentially, both the view (UI elements) and the state (instance variables of the various classes) need to be considered. The components involved are the VolcanoCards, the caves, the dragon tokens, the chit cards, the order of the players and which player's turn it is.

Level of difficulty

Extreme. This was difficult to implement, but the reason is less related to the extensibility of our design. At a high level, the game allows the players to start a new game (with randomised VolcanoRing, ChitCard and starting cave configurations) or load an existing one. On one hand, it was easy to define a LoadSave interface exclusively for classes involved in the saving/loading functionality using the load() and save() methods (Interface Segregation Principle), and we allow the main application class to track an array list of LoadSave instances. These LoadSave instances are 'registered' into the main class array, and whenever the game is loaded, the application polymorphically calls on its LoadSave instances to load their state. This means that the specific class implementing the LoadSave interface is inferred at runtime, and its specific implementation of load() is called (Liskov Substitution Principle). Then, we just needed to design a simple interface (LoadSaveUI) to display slots (maximum 8 in our case) for players to save to or load from.

So, the extensibility of our design was not an issue. One of the reasons why it was difficult was because we could not make use of FXGL's built-in loading and saving functionality, as it used binary storage of information, which was disallowed. This meant that we had to define our own saving/loading scheme, which was different for each class, since different classes stored different state information. On top of that, some difficulty arises from the fact that our system intelligence was distributed fairly sparsely, since we had classes for model, controller, view, entity factory, etc. During implementation, we had to consider how the different components fit together without breaking our existing design, and this was somewhat limited by our chosen development framework (FXGL). It was important for us to realise that the initGame() method (overridden method from FXGL's GameApplication class) was the entry point for us to implement separate logic for starting a new game versus loading a saved game.

Reflections on Sprint 3 Design

What worked well to support the implementation of this extension:

- We previously defined model classes to store the information on the state of the board components, and we stored a collection of these instances in an array on creation. Whenever the factory classes spawned the entities, they were populated in their respective arrays (e.g. VolcanoRingFactory populates the volcano ring array) By maintaining references to these VolcanoCards, regardless of any change to the states of the instances throughout the game's progression, we are guaranteed to always have the latest state ready to be saved. In particular, we can directly iterate through these arrays to store the state into a save file.
- In Sprint 3, having a main and game menu was not explicitly required, but we included it in our submission. This extra effort paid off as we could reuse these 'menu' components to implement the save(in-game menu) and load(on the main menu) functionalities. Specifically, we could easily add buttons for these two functions, and provide the implementations for the callback functions.

Design deficits:

- In Sprint 3, caves were static, and so it was reasonable to just have a CaveFactory class to spawn the caves once, and ignore them for the rest of the game (no need to maintain state). However, in implementing the save/load functionality, we realised that defining an explicit Cave class was necessary since loading the game involves spawning caves which needs:
    - The animal on the cave
    - The position of the cave (x,y coordinates)
    - The ID of the player it corresponds to
- A similar case can be said for the board components that have dynamic UI components. We did not store the x, y coordinates in the Model classes previously, and consequently necessitated refactoring of these classes to include getters and setters for these attributes.

3.3 Self-Defined Extension 3 - Equality Boost

Description

The "Equality Boost" extension adds a feature to the game where if a player consistently reveals incorrect cards three times in a row, they receive an "Equality Boost" card. This boost allows them to automatically move one step forward on their next turn. However, if the next card is occupied, the player's position will not move. This feature aims to balance the gameplay and offer a second chance to struggling players.
Human Value: Equality under Universalism in Schwartz's Theory of Basic Human Values [2]

Level of difficulty

The level of difficulty in implementing this extension is moderate.

Factors contributing to difficulty

Distribution of System Intelligence:
Player Class: The intelligence for tracking incorrect reveals and managing the "Equality Boost" is centralised in the Player class. This involves maintaining counters and state flags that track the player's performance and boost status.
FieryDragonsApplication Class: The main application class had to be adjusted to handle new game logic for incorrect reveals, awarding boosts, and using boosts effectively.

Code Smells:
Complex Conditional Logic: The logic for handling incorrect reveals and boosts adds conditional complexity to the handleCircleClick method, which could be prone to errors if not carefully managed.

Design Patterns Used:
Observer Pattern: The interaction between different components (e.g., player turn manager, chit card adapter, and text display manager) follows a pattern where changes in one component (e.g., player's state) trigger updates in others.
Factory Pattern: The game heavily relies on factory classes for spawning game entities, and the new feature had to integrate seamlessly with these existing factories.

Reflection on what made this extension easy/difficult to address

Easy Aspects:

Modular Design: The game's design, which uses factory patterns and separated responsibilities (e.g., player state management, UI updates), made it straightforward to add new features without deeply entangling with existing logic.
Centralised State Management: Managing the incorrect reveal counter and equality boost state within the Player class provided a clear and contained way to implement the feature.

Difficult Aspects:

Conditional Logic Complexity: Adding the equality boost logic introduced new conditional checks that increased the complexity of the handleCircleClick method. Ensuring that all conditions (e.g., incorrect reveals, end of turn, equality boost activation) were correctly handled required careful thought and testing.
State Synchronisation: Keeping track of the player's state across different parts of the game (e.g., UI updates, game logic) required meticulous updates to ensure that the player's incorrect reveal counter and equality boost status were accurately reflected in the gameplay.

3.4 Self-Defined Extension 4 - Shop with purchasable items

Description

The shop function in the game is a feature that allows players to purchase effects using a point system that updates based on the player making the right move. This function enhances the gameplay experience by providing strategic options as well as building their intelligence to plan out how to distribute their points to their advantage based on the current game situation. This is related to the "intelligent' human value under Achievement in Schwartz's Theory of Basic Values. [2]

Level of difficulty

Moderate as it is just needed to add on further in the base code from what is already implemented.

Factors contributing to difficulty

Distribution of system intelligence:

The UI was enhanced to provide real-time feedback on transactions, requiring the implementation of asynchronous communication mechanisms to keep the UI synchronised with the server's state. These changes ensure that the shop function operates securely and efficiently, providing a seamless experience for the player. These changes were made in TextDisplayManager.

Code smells:

One of the concerns is the growing complexity of the handleCircleClick method, which now handles multiple responsibilities, including chit-card uncovering, dragon pirate card handling, and token movement validation.

Design patterns:

The Factory pattern is utilised in the creation and management of various shop items and inventory through a centralised factory class, ensuring that item creation is consistent and easily extendable. The Singleton pattern is employed in the ShopFactory class, which oversees the entire shop operation, ensuring that a single instance controls the shop's state and interactions. Additionally, the Command pattern is subtly present in handling player purchases and inventory updates, encapsulating these operations as objects to decouple the request sender from the logic that handles the request.

4. Retrospective on Sprint 3 Design/Implementation

Changes To Make if Starting Over

Modularize Further: Increase the use of modularization to break down complex logic into smaller, more manageable components.
Unified State Management: Implement a more centralised state management system to streamline the saving/loading process.
Design Patterns: Apply more design patterns (e.g., Strategy Pattern) to handle varying behaviours more cleanly and reduce conditional logic complexity.

How These Changes Would Ease the Incorporation of Extensions

Modularization: Smaller, focused modules would make it easier to understand and extend specific functionalities without affecting other parts of the system.
Centralised State Management: Simplifying state tracking and updates would reduce the risk of inconsistencies and errors during state transitions.
Design Patterns: Using appropriate design patterns would enhance code readability, reduce code smells, and make it easier to implement new features with minimal impact on existing code.

MONASH University

5. Future Strategies and Techniques

This section discusses future strategies and techniques based on our reflection on areas for improvement. We suggest some techniques, and outline the evaluation process, which often involves group reviews, or by reflecting on written code.

The "code first, think later" mentality tends to accumulate technical design debt, and needs to be changed.

Often, we are focused on implementing logic, leading to messy, cluttered code. We may take shortcuts to give a working solution, with little to no regard for maintainability

1) Techniques moving forward
    a) Fit the problem to known design patterns. For example, the MVC pattern is reusable, and easily applied to enforce separation of concerns in applications. This was very applicable to our FieryDragons game.
    b) Practice modularisation: think of how a huge task, e.g. flipping of chit cards and their effects can be decomposed into smaller logical tasks: flip animation, validation, dragon token animation, updating the state of classes involved in this interaction
2) Evaluation
    a) Conduct code reviews frequently with others. During the development process, the developer will often have a better understanding of the code, which could lead to a biased assessment of the design.
    b) Having fresh perspectives on the design would provide a fairer assessment, so evaluation should be carried out as a group.

Magic numbers should be eliminated

Without context, it becomes difficult to infer their semantics. Again, this could be a by-product of the "code first, think later" mentality. This problem is only exacerbated with time.

1) Techniques moving forward
    a) Good practices for constants: should be defined as instance-level attributes in the class.
    b) Avoid scattering them throughout methods as local variables
    c) If they are used application-wide (like in the case of our FieryDragons game) consider moving them to a specific file.
2) Evaluation
    d) Evaluation is straightforward by scanning the code for ambiguous numbers and refactoring them.
    e) However, rather than doing this as the final step, to avoid difficulty with manually tracking application-wide usage, it is often better to define them as early as possible.

Code documentation is second to quality design

This is not only for the benefit of others who may continue our work but also for ourselves in the future.

1) Techniques moving forward
    a) Write future-proof documentation. Don't assume knowledge, and instead imagine ourselves in a few months in the future. If the semantics would be lost by then, it indicates a lack of clarity

b) Proper naming conventions of variables and descriptive names are inherently part of the documentation. With proper naming, redundant documentation can be avoided. The tradeoff of the initial effort required to come up with names that capture the semantics would be beneficial in the long run.

2) Evaluation
   a) Feedback from others. From our first-hand experience throughout these Sprints, teammates often struggle to understand other's documentation (or sometimes a lack of it), and this is an indicator of poor documentation standards.
   b) When scanning through variable names, identify short variable names. Often, single-character variables like 'a', 'b','x', etc. are problematic and should be refactored.

6. Executable Instructions

Instructions:
The executables were built using Maven based on the pom.xml file. The output is a zip file and it should be located in a directory called 'target'.

You should be using Java SDK version 22.
To produce the zip file: In your IDE, run the command mvn javafx:jlink using Maven.
In IntelliJ, this can be done by pressing the Ctrl key twice and then searching for this command.

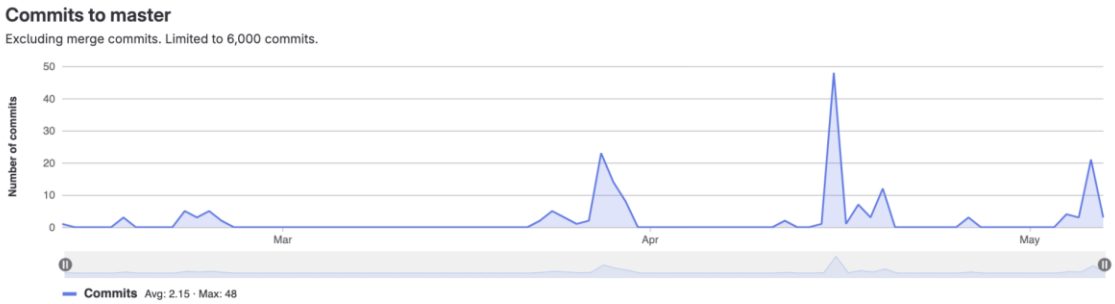
Detailed instructions can be found in the README section.

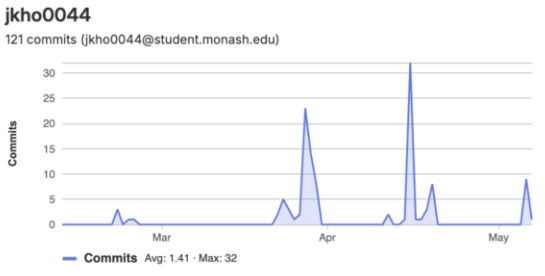
7. Sprint Contributions

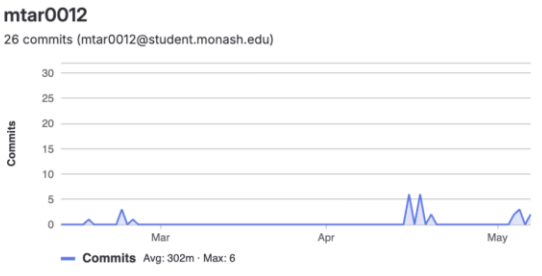
Wiki GitLab Link

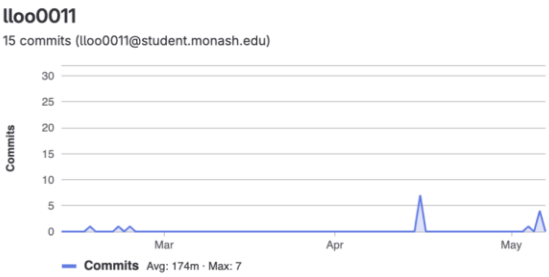
https://git.infotech.monash.edu/FIT3077/fit3077-s1-2024/MA_Tuesday08am_Team123/-/wikis/Contribution-log

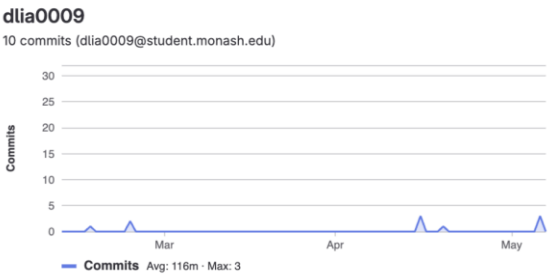
Contributor Analytics











MONASH University

8. References

[1] https://edstem.org/au/courses/14452/discussion/1994103
[2] S. H. Schwartz et al., “Refining the theory of basic individual values,” J. Pers. Soc. Psychol., vol. 103, no. 4, pp. 663–688, 2012