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

MONASH University

2. Object-Oriented Class Design

Notes:

1) The new sections on the class diagram are in red.
2) Methods are added whenever possible, however, to avoid cluttering the diagram, some are omitted. This is also because some methods are from Sprint 3, and will not be repeated.
3) As per previous feedback, the composition/association relationship arrows now end with an arrow.
4) 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 own implementation.

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

General challenges and successes
Overview of the extensions implemented

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.

Reflections on Sprint 3 Design

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 that has its own 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.

Level of difficulty

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.

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 (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.

Draft:

Reflection on Sprint 3 Design

In Sprint 3, our team undertook a comprehensive review and evaluation of each member's submission from Sprint 2. We collectively decided which parts from each submission would serve as the basis for our combined solution, ultimately leading to the development of a fully functional Fiery Dragon's game without any extensions. As we moved into Sprint 4, we were tasked with implementing extensions to our Sprint 3 design, providing valuable insights into the strengths and weaknesses of our initial implementation.

Description of Sprint 3

Sprint 3 involved creating a baseline version of the Fiery Dragon's game. The core functionalities included player movement, token management, and basic game mechanics without any additional extensions. The design emphasises modularity and separation of concerns, with key classes like Player, DragonToken, and various factories (VolcanoRingFactory, ChitCardFactory, etc.) handling specific responsibilities.

Reflection on Extension 1: New Dragon Card 1

Sprint 4 Description: The token moves backwards until it finds a free cave without losing the player's turn.

Level of Difficulty:

- **Difficulty:** Moderate. The existing movement logic needed enhancement to support backward movements and validate free caves.

Sprint 3 Design Impact:

- **Challenges:**

    - Our design had to be modified to handle reverse movements.
    - We had to introduce additional checks for free cave validation.
- **Strengths:**
    - The modular structure of our DragonToken class facilitated the addition of new movement methods.
    - The existing turn management system was flexible enough to incorporate the continuation of the player's turn.

Design Principles and Patterns:

- **Single Responsibility Principle:** The DragonToken class had a clear responsibility for managing token state and movement, which made it easier to add new movement methods.
- **Open/Closed Principle:** The modular design allowed for extensions without modifying existing code significantly.
- **Strategy Pattern:** If applied, this pattern could have helped in dynamically selecting different movement strategies (forward, backward) for tokens.

Lessons Learned: If we were to revisit our Sprint 3 design, we would implement a more flexible movement system from the outset, capable of handling both forward and backward movements. This would have made the integration of this extension simpler and less error-prone.

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.

Reflections on Sprint 3 Design

What worked well to support the implementation of this extension:

- We previously defined model classes to store the information of 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 menu and game menu was not explicitly required, but we included it as part of our submission. This extra effort paid off as we could reuse these 'menu' components to implement the save(in-game menu) and load(on 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.

Level of difficulty

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.

Draft:

Reflection on Extension 2: New Dragon Card 2

Sprint 4 Description: The token swaps position with the closest token and the player loses their turn.

Level of Difficulty:

- **Difficulty:** High. Complex logic was needed to ensure valid swaps, handle cave constraints, and manage cyclical board positions.

Sprint 3 Design Impact:

- **Challenges:**

    - The swap logic required extensive case handling to ensure correctness.
    - The cyclical board added complexity to the distance calculations.
- **Strengths:**
    - The use of a DragonTokenManager to track token positions streamlined the process of finding the closest token.
    - Our modular design allowed for easier incorporation of the swap logic within existing game mechanics.

Design Principles and Patterns:

- **Single Responsibility Principle:** The DragonTokenManager was solely responsible for managing and calculating token positions, aiding in the implementation of the swap logic.
- **Open/Closed Principle:** While the initial design allowed for extension, the swap logic revealed areas where more flexibility was needed.
- **Strategy Pattern:** This pattern could have been beneficial for handling different swap strategies and distance calculations.

Lessons Learned: To make this extension easier to integrate, we would have designed our token movement and distance calculation logic to be more adaptable to various movement scenarios. Implementing a more robust token management system in Sprint 3 would have reduced the complexity of adding such extensions.

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 and still take their turn normally. This feature aims to balance the gameplay and offer a second chance to struggling players. This is related to ‘Equality’ under Universalism in Schwartz’s Theory of Basic Values.

Level of difficulty

The level of difficulty in implementing this extension is moderate.

Factors contributing to difficulty (e.g., distribution of system intelligence, code smells, design patterns used)

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.
State Management: Introducing new state variables like incorrectRevealCounter and hasEqualityBoost increases the risk of state management issues if not properly initialised and updated.

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 clearly 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

Description

Factors contributing to difficulty (e.g., distribution of system intelligence, code smells, design patterns used)
Reflections on Sprint 3 Design

Level of difficulty

3.2 Constraints Introduced by Sprint 3 Design/Implementation

General constraints identified
Specific constraints for each extension

3.3 Retrospective on Sprint 3 Design/Implementation

Changes you would make if starting over
How these changes would ease the incorporation of extensions

3.5 Future Strategies and Techniques

Strategies and techniques to improve future practice
Thought process on how they might work in similar situations

Bad habits that need to be changed:

1) Code first, think later. Accumulating technical design debt
    a) Often engrossed with implementing logic, leading to messy cluttered code.
    b) We may take shortcuts to give a working solution, with little to no regard for maintainability
    c) Techniques moving forward:
        i) Fit the problem to known design patterns. For example, the MVC pattern alone is usually small and reusable, easily applied to enforce separation of concerns
        ii) Strategy: 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 state of classes involved in this interaction
        iii) Evaluation:

2) Magic numbers (code smell).
    a) Without context, it becomes difficult to infer their semantics
    b) Problem exacerbated with time
    c) Techniques moving forward:
        i) Good practices for constants: should be defined as instance-level attributes, and not scattered throughout methods (or even classes) as local variables.

3) Quality of code documentation
    a) Not only for others who may continue our work, but also for ourselves in the future
    b) Techniques moving forward:
        i) To write future-proof documentation: Don’t assume knowledge. Put yourself in the shoes of yourself a few months in the future. Would you still be able to infer the semantics? If not, more detail is required.
        ii) Feedback from others also helps. If they struggle to understand the documentation, it is likely lacking in clarity

MONASH University

4. 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.

5. Sprint Contributions

Wiki GitLab Link

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

Contributor Analytics

6. References

[1] https://edstem.org/au/courses/14452/discussion/1994103