FIT3077 Software Engineering: Architecture and Design

Sprint Four (20%) Specifications

Group: MA_Wednesday04pm_Team888

Members: Arvind Siva, Lee Sing Yuan, Mohamed Areeb Ilham Riluwan, Tharani Prathaban

Due date: Thursday, 6 June 2024

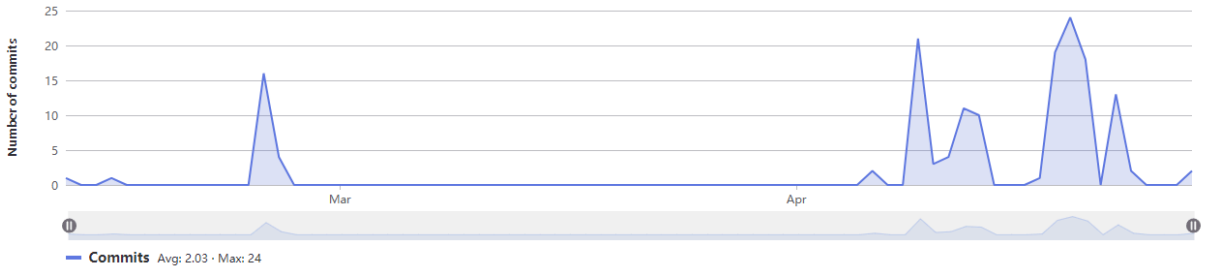
Company: Nexus DevOps

Table of Contents

Contributor analytics

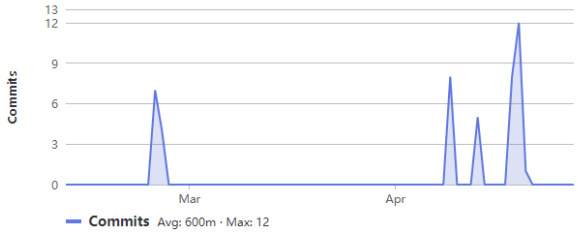
Commits to master

Excluding merge commits. Limited to 6,000 commits.



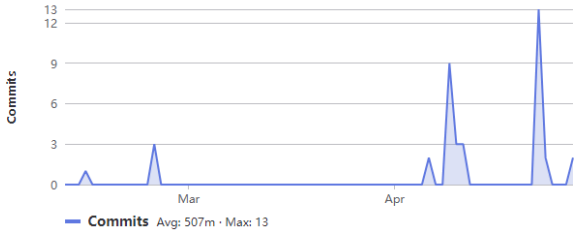
Lee Sing Yuan

45 commits (slee0163@student.monash.edu)



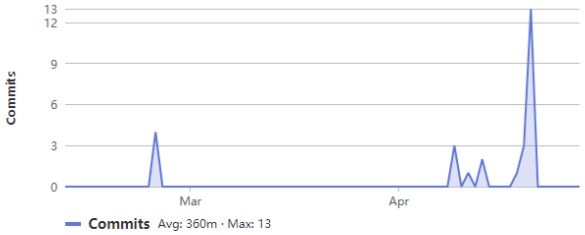
Arvind Siva

38 commits (asiv0014@student.monash.edu)



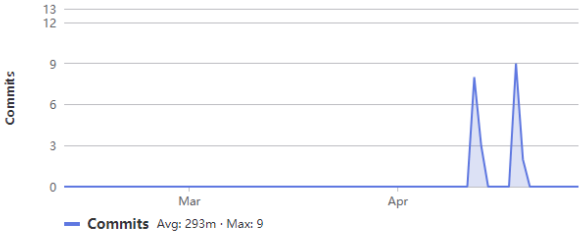
milh0002

27 commits (milh0002@student.monash.edu)



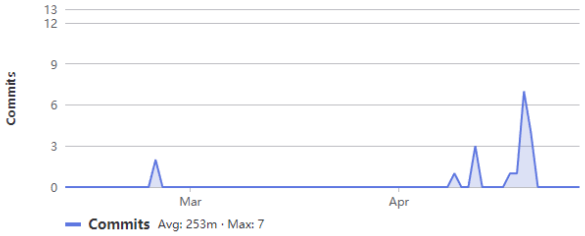
Arvind Siva

22 commits (arvindsiva11@gmail.com)



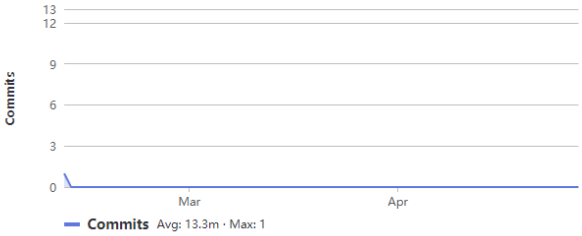
tpra0011

19 commits (tpra0011@student.monash.edu)

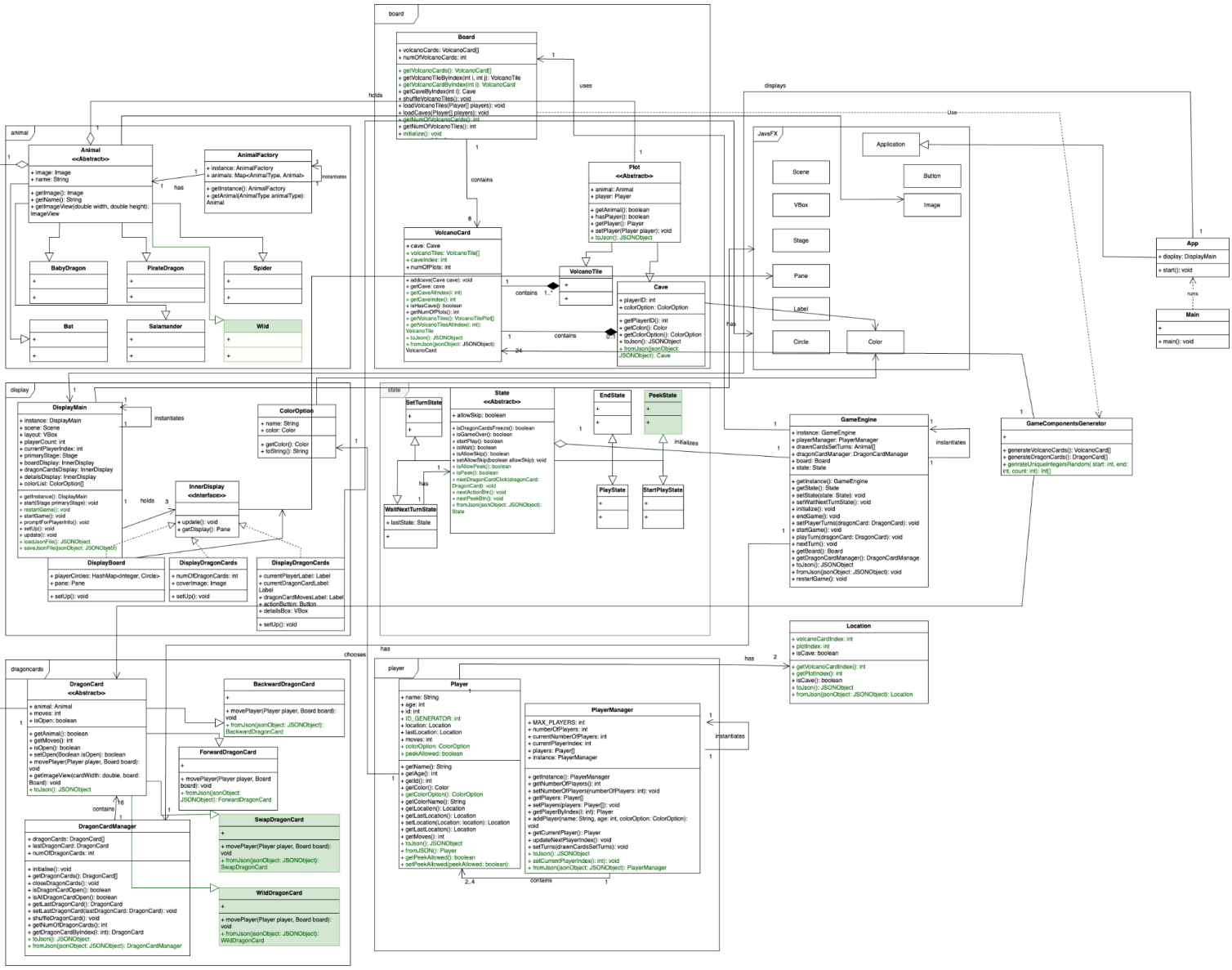


Matt Chen

1 commit (matt.chen@monash.edu)



Sprint 4 UML Class Diagram



If the UML diagrams are not clearly visible, please specify that they can be found in the **documents** folder in the repository for better clarity.

Brief Introduction

In Sprint 3, our primary goal was to create a robust design that would facilitate the seamless integration of future extensions. Reflecting on this process, we as a team, found incorporating the extensions to be both challenging and insightful. Below, we as a team, individually discuss the level of difficulty for each extension implemented by each of us, the impact of our Sprint design, and strategies for improving future practices.

Extension 1: Create new swap place dragon card [Requirement]

Implemented by Lee Sing Yuan

Difficulty Level: 2/5 [on a scale of 1 to 5 where 1 is easy and 5 is difficult]

Discussion:

Challenges:

There were several challenges, primarily focused on determining how to calculate the distance between players, when teleportation should be allowed, and whether teleportation to another player is possible while they are in the caves. Addressing these questions also required identifying the best methods to implement. For instance, to check if the nearest player is in a cave, it is important to place this check in a way that ensures optimal code quality. I decided to override the method from the Dragon Card abstract parent, as this approach aligns best with the current code design and ensures adherence to encapsulation and SOLID principles. An alternative was to perform the checks within the state classes, but this would violate the Single Responsibility Principle (SRP) for states, as they would have to manage both the game state and the validity of moves.

Design and Implementation Factors:

Starting from sprint 3, we decided to utilise a singleton class to manage all the players. This made it easier to encapsulate the functionality of calculating the distances between players, as we could simply call the PlayerManager class to retrieve the list of players. Additionally, since we have another singleton class to manage all the dragon cards, adding a new dragon card became straightforward: we only needed to create the new dragon card class and add it to the singleton. Besides that, there was also an application of the bridge structural pattern which is accomplished by splitting a major class into separate hierarchies such as the DragonCard abstract class. This reduced the coupling between classes and made it easier to implement the new swap dragon card feature. Despite the aforementioned challenges, implementing this feature was relatively easy. However, there are several improvements that could be made to further enhance the code quality, as discussed below.

Future Improvements:

Code quality:

The code is tightly coupled and could benefit from further encapsulation to enhance extensibility. For instance, creating an Action class to be used with the dragon cards would help achieve this. The current use of state in the code is commendable, but it could be improved by implementing a Facade. For example, to introduce a new feature like a dragon card that appears randomly, a Facade could be used to update the state seamlessly. Additionally, an abstract factory could be created to instantiate each animal's dragon and volcano cards. This approach would further encapsulate the code, allowing the GameComponentsGenerator to simply call the factory method to produce all the necessary cards.

User experience:

Additionally, we could customise the image for the swap dragon card to enhance the user experience. Finally, implementing animations or other forms of feedback to indicate which players are swapping would further improve the overall user experience.

Extension 2: Saving and Reloading The Game [Requirement]

Implemented by Arvind Siva

Difficulty Level: 2/5 [on a scale of 1 to 5 where 1 is easy and 5 is difficult]

Discussion:

Challenges: The main challenge I faced was to select a file format to save the data required for gameplay and reload them correctly. I selected the JSON file format that can save all the required data of the gameplay and be easily read to recreate all the game objects and reload the game. Overall the file format was the main challenge.

Design and Implementation Factors: All the crucial game components to run the game such as the Animal, DragonCard, VolcanoCard, Cave, and VolcanoTile had inheritance or abstraction. This made it easy to create methods in the parent class to generate and read a JSON representing the current state of the object and the child classes could override the methods for a more specific implementation. Moreover, critical game components like the DragonCard and Plot (inherited by VolcanoTile and Cave) had integrated bridge design patterns with Animal. This made it easy to create and read JSON files for them because there are fewer classes because the bridge pattern allows them to mix and match. This allows me to create a JSON format with fewer key-value pairs that can be read faster. Then the game has a State class which uses the State design pattern to assist the GUI handled by the display package to decide the next action based on user interaction. This allows me to create a JSON that represents the current state of the gameplay which makes it easy to save and reload the game exactly where at the gameplay it was saved. The GameEngine class that controls the game logic is not a God class by splitting some logic responsibilities into other classes such as DragonCard, DragonCardManager, PlayerManager, Board, and State. GameEngine makes use of Mediator and Singleton design patterns. Since the game engine is not a god class, I easily created a JSON representing the GameEngine and within it, I have JSON objects representing the classes that handle the split responsibilities. This is the final JSON file saved and reloaded to start the game. This design is robust and efficient as the JSON format is easily readable and compact which makes it fast to save and reload the game correctly. The current design of the GameEngine made it easy to reset all the game components and the state to restart the game. Earlier the VolcanoCards were designed to hold multiple VolcanoTiles and Caves which made it easy to create VolcanoCards of different sizes with Caves in different positions with a few tweaks. Moreover, the DisplayBoard class was designed to render VolcanoCards of any configuration size which made it easy to display any number of VolcanoCards without any change to the UI and the logic for player movement already took this into account earlier which does not have to be modified.

Future Improvements: The current design has many classes to deal with when saving the current gameplay as a JSON to be reloaded which makes it harder to maintain. In the future, I would explore other design patterns to further simplify the existing code which will make extending this feature much easier and less time-consuming. The current design involves button interaction to save and reload the game in the future with more time we could explore on automating this to make the user experience smoother. The current design when saving JSON for VolcanoCards saves every single data about it which is inefficient if 2 or more VolcanoCards have the same configuration. In the future, we could explore ways to reduce this data duplication by using count key values to describe the number of such duplicated VolcanoCards instead of duplicating them in the JSON file.

Extension 3: Wild Card (self-defined)

Implemented by Tharani

Feature explanation:

The Wild Card feature adds a special twist to the game with the introduction of the WildDragonCard. This card gives players a break from the usual rules, allowing them to move forward on the board without having to have had flipped a dragon card that matches the animal of the land they're on. It's designed to shake things up and keep players on their toes. When a player flips a Wild Card, they move a specified number of spaces forward, regardless of what animal is on the land they are currently on, which can lead to interesting interactions with VolcanoCards, Plots, and Caves. The idea is to inject more dynamic movement options and add strategic depth to the game. The Wild Card brings an element of unpredictability and adaptability, pushing players to think strategically and creatively. What I love about the Wild Card is how it encourages players to explore new strategies and adapt to whatever the game throws at them. It breaks the routine and adds a fresh challenge, keeping the game exciting and engaging. This aligns perfectly with the human value of Stimulation, as it keeps players intellectually engaged and challenges them to strategically leverage off the advantages the Wild Card offers. The excitement and novelty it brings make every game session more thrilling and fun.

Difficulty Level: 1/5 [on a scale of 1 to 5 where 1 is easy and 5 is difficult]

Discussion:

Challenges: One of the main challenges I ran into was making sure the Wild Card fit seamlessly with the existing game mechanics. I had to harmonise its movement with various elements like Volcano Cards, Plots, and Caves. This meant really diving into the game's internal logic to avoid any disruptions or inconsistencies. Figuring out how the Wild Card should interact with other elements, especially Volcano Cards and Caves, added another layer of complexity.I spent quite a bit of time deciding whether the Wild Card should allow a player to enter the game immediately after flipping it, or only benefit them once they'd entered the cave by flipping the correct dragon card. The first option made more sense for a Wild Card and was luckily less complicated because it required fewer scenario validations. Handling transitions between different card types and managing edge cases like overlapping player positions or unexpected card combinations was tricky and took some time to get right. Debugging and testing these interactions were crucial to ensure the Wild Card behaved as expected. This involved a lot of time-consuming work to identify and fix issues related to movement mechanics, player positions, and card interactions. Making sure everything worked across various game scenarios required extensive testing and iteration. On top of that, I had to ensure backward compatibility with existing game features, which added another layer of complexity to the testing process. Lastly, balancing the novelty and unpredictability of the Wild Card with the overall user experience was a nuanced challenge. The goal was to stimulate strategic thinking and creativity without overwhelming players with too much complexity or randomness. Finding the right balance

between making the gameplay exciting and maintaining a cohesive user experience required a lot of careful consideration and tweaking.

Design and Implementation Factors: In designing and implementing the Wild Card feature, I built on the foundational design principles and patterns we established in Sprint 3 to make sure it was both robust and flexible. I really focused on encapsulation and the Single Responsibility Principle (SRP) when designing the WildDragonCard class, which extends from the DragonCard abstract class we introduced in Sprint 3. This helped the WildDragonCard class stay focused on its main job: handling forward movement logic, which makes the code easier to maintain and understand. Using the Factory Pattern in the AnimalFactory class from Sprint 3 was a big help too. It allowed me to create instances of Wild type animals dynamically, which is great for keeping the code modular and easy to extend. This way, adding new animal types or card variants in the future will be a breeze. Polymorphism played a key role here as well. By inheriting from the DragonCard class and overriding the movePlayer method, the WildDragonCard could provide its specific movement behaviour while fitting right into the existing game structure. The bridge pattern between DragonCard and WildDragonCard was crucial in this regard, allowing the WildDragonCard to leverage the existing DragonCard framework while introducing its unique features seamlessly. I also made sure to stick to the Don't Repeat Yourself (DRY) principle by using the existing AnimalFactory class from Sprint 3 design to handle Wild type animal creation in one place. This reduces redundancy and the chance of errors from duplicated code. This cohesive design approach, which we set up in Sprint 3, really made the development of the Wild Card feature smooth and effective.

Future Improvements: When I think about future improvements for the Wild Card feature, a few ideas come to mind that could really boost its functionality and maintainability. First off, extracting common behaviours into helper methods or classes would streamline the codebase and make it easier to tweak or update down the line. For example, when I implemented the movePlayer method in the WildDragonCard, I noticed it was pretty similar to the one in the ForwardDragonCard, just without the animal matching checks. If we had predefined methods or classes for these common movement types, it would’ve saved a ton of time and effort. This is definitely something to consider for future updates. I also want to make sure the design aligns better with the Open/Closed Principle to keep the system more extensible. This means making it easier to add new Wild Card behaviours without messing with the existing code, so we can introduce new movement types or interactions seamlessly. A strategy pattern for handling different movement behaviours could also help a lot. By decoupling the movement algorithms from the Wild Card class, we’d be able to add or change movement strategies without touching the existing code. On top of that, I’m planning to add comprehensive unit and integration tests specifically for the Wild Card feature. This proactive approach to testing will help ensure consistent behaviour across different game scenarios, making debugging and maintenance a lot smoother.

Extension 4: Peek at All DragonCards (self-defined)

Implemented by Areeb

Feature explanation:

In the game Fiery Dragons, each player has a unique opportunity to activate a special feature by pressing the "Peek" button located in the top right corner of the screen. When this button is pressed, all dragon cards on the game board are temporarily flipped over, revealing their faces. This allows players to see and memorise the positions of all the cards.

During this peeking phase, the cards remain visible until the player decides to end the peeking session by pressing the "Stop Peeking" button. This feature is designed to assist players in remembering the locations of the dragon cards, making it easier for them to make accurate guesses and matches throughout the game. This memory-enhancing mechanism adds a strategic element, helping players to plan their moves more effectively and increase their chances of winning.

Difficulty Level: 2/5 [on a scale of 1 to 5 where 1 is easy and 5 is difficult]

Discussion:

Challenges: Implementing the "Peek at All DragonCards" feature in the Fiery Dragons game presented several challenges. Managing the game state during the transition between normal gameplay and the peek phase required careful consideration to ensure consistency and prevent disruptions. Creating an intuitive and responsive user interface for the peek functionality was also challenging, particularly in maintaining a seamless experience for players across different devices. Additionally, ensuring the game remained performant during the peek phase, given that all cards on the board are revealed simultaneously, posed potential resource consumption issues.

Design and Implementation Factors: The design and implementation of the peek feature involved several key principles and patterns. The State Design Pattern was pivotal in managing the different states of the game, such as transitioning from a normal state to a peek state and back. This pattern helped encapsulate state-specific behaviours and made the state transitions smooth and scalable. For instance, the PeekState class effectively uses this pattern by implementing specific behaviours for the peek phase and providing methods to transition back to the previous state. This was made possible after the sprint 3 design decisions to include states.

The Observer Pattern was utilised to notify and update different components of the game when a state change occurs, such as starting or stopping the peek phase, ensuring that the game's UI remained in sync with the game's state. Additionally, the Command Pattern was used for implementing the actions associated with the "Peek" and "Stop Peeking" buttons, allowing for an easy way to extend the game with new features without heavily modifying the existing codebase. This is evident in the nextPeekBtn() method, which handles the transition out of the peek state.

Incorporating human values into the design was crucial. The peek feature promotes **fairness and equity** by levelling the playing field for all players, regardless of their memory skills. It stimulates **mental engagement and achievement** by challenging players to remember card positions, thereby enhancing their strategic thinking. The feature also supports **user autonomy and independence** by allowing players to choose when to use it, giving them control over their gameplay. Furthermore, it encourages **benevolence and social interaction** in multiplayer settings, fostering cooperation and shared strategies.

Future Improvements: To enhance the "Peek at All DragonCards" feature, several improvements could be considered. Enhancing the user interface to be more adaptive to different screen sizes and orientations would improve accessibility and user experience. Implementing advanced rendering and caching techniques could handle the increased load during the peek phase without compromising performance. Introducing a setting that allows players to configure the duration of the peek phase would accommodate various skill levels and strategies. Additionally, providing players with visual or auditory feedback when the peek phase starts and ends would enhance the game's interactivity and user engagement.

Description of executable

The executable currently in the repository was created from a windows' Operating System. If you would like to run it on other platforms, Please rebuild the artefact following the steps in the Readme.md. The instructions to build the artefact is also present in the Readme.md

It could be summarise like this:

1. Clone the project on your local machine
2. Open the project in Intellij
3. Wait for Maven and all the project dependencies to set up
4. In Intellij, navigate to File > Project Structure > Artefacts
5. Press the add button and select JAR
6. Select Main as the main class
7. Press OK
8. Press Apply, then press OK
9. In Intellij, navigate to Build > Build Artefacts
10. Press Build for FieryDragons:jar