FIT 3077 Software Engineering: Architecture and Design

Sprint Three Documentation

Group Name: Stack Underflow

(MA_Wednesday04pm_Team128)

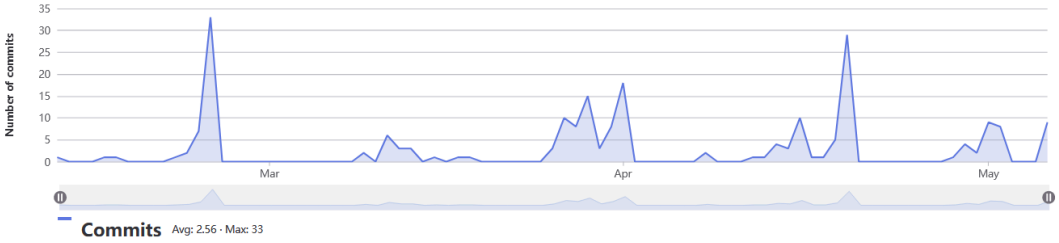
Members :

Chang Yi Zhong (33991499)
Nikhita Peswani (31361552)
Lim Hung Xuan (33984972)
Enrico Tanvy (33641668)

Contribution analytics

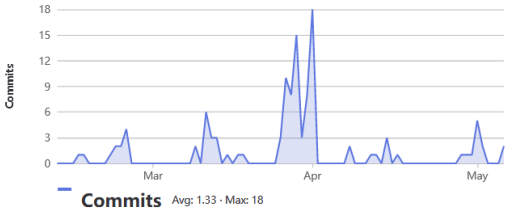
Commits to master

Excluding merge commits. Limited to 6,000 commits.



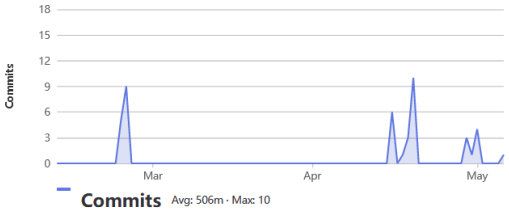
Chang Yi Zhong

113 commits (ycha0154@student.monash.edu)



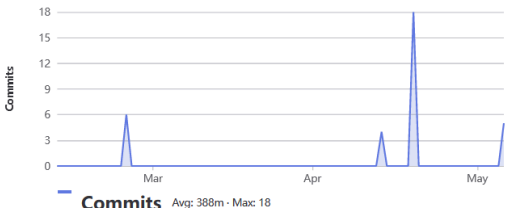
hlim0069

43 commits (hlim0069@student.monash.edu)



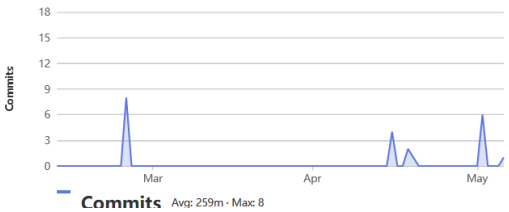
npes0001

33 commits (npes0001@student.monash.edu)



enri0002

22 commits (enri0002@student.monash.edu)



Description of self-defined extensions

1. **New behaviour when token attempts to move to an occupied tile**

Instead of the token not moving at all if the target tile is occupied, the tokens will instead swap places. This means that if Player 1 flips a chit card that allows it to move 3 tiles ahead, but Player 2 is already occupying that specific tile, Player 1 will swap places with Player 2. This means, Player 2 will move 3 tiles back to Player 1's original tile, and Player 1 will move 3 spaces forward. However, Player 1 will lose their turn, and the turn is changed to the next player.

If Player 1 is still in their cave, and flips a chit card that allows them to land on an occupied tile, Player 1 will stay in their cave and their turn will end.

If Player 1 is already out of their cave, but gets unlucky and flips a pirate dragon card, which makes them move backwards 2 tiles, and the target tile is already occupied by Player 2. Player 1 will swap places with Player 2, meaning Player 1 will move 2 spaces backwards and Player 2 will move 2 spaces forwards, giving Player 2 an advantage.

This essentially adds more skill expression to the game. This extension rewards players for flipping the correct chit card that allows them to exactly land on a tile that is occupied by a player. This also adds another aspect to the game, the current player has to weigh the pros and cons of landing on an occupied tile, as even though they get to swap places with the other player, essentially dragging them back and hindering their progress; the current player needs to consider the fact that they will lose their turn as by doing so, this adds more depth and decision-making into the overall gameplay.

2. **Allowing the player to choose the number of volcano cards before the start of the game**

In addition to letting the player choose the number of players before the start of each game, we made a new UI that allows the player to choose the number of volcano cards they would like to have, each volcano card has 3 tiles, as per the game rule.

This allows players to customise the size of the game board by choosing the number of volcano cards before starting each game. This new feature adds a layer of strategy and customization, allowing players to tailor their gaming experience. We have set the minimum number of volcano cards to 8 (24 tiles), in line with the original game rule, and capped it at 21 (63 tiles). Through testing, we discovered that having more than 21 volcano cards made the game visually overwhelming and excessively prolonged each session, diminishing the overall experience. This customization

option achieves a balance, providing players with flexibility while maintaining gameplay quality.

Schwartz's Theory of Basic Human Values
Schwartz's Theory of Basic Human Values identifies ten broad values, which are universal across cultures and help to explain human behaviour. Among these values, both our self-defined extensions particularly address Stimulation, Achievement, and Self-Direction.

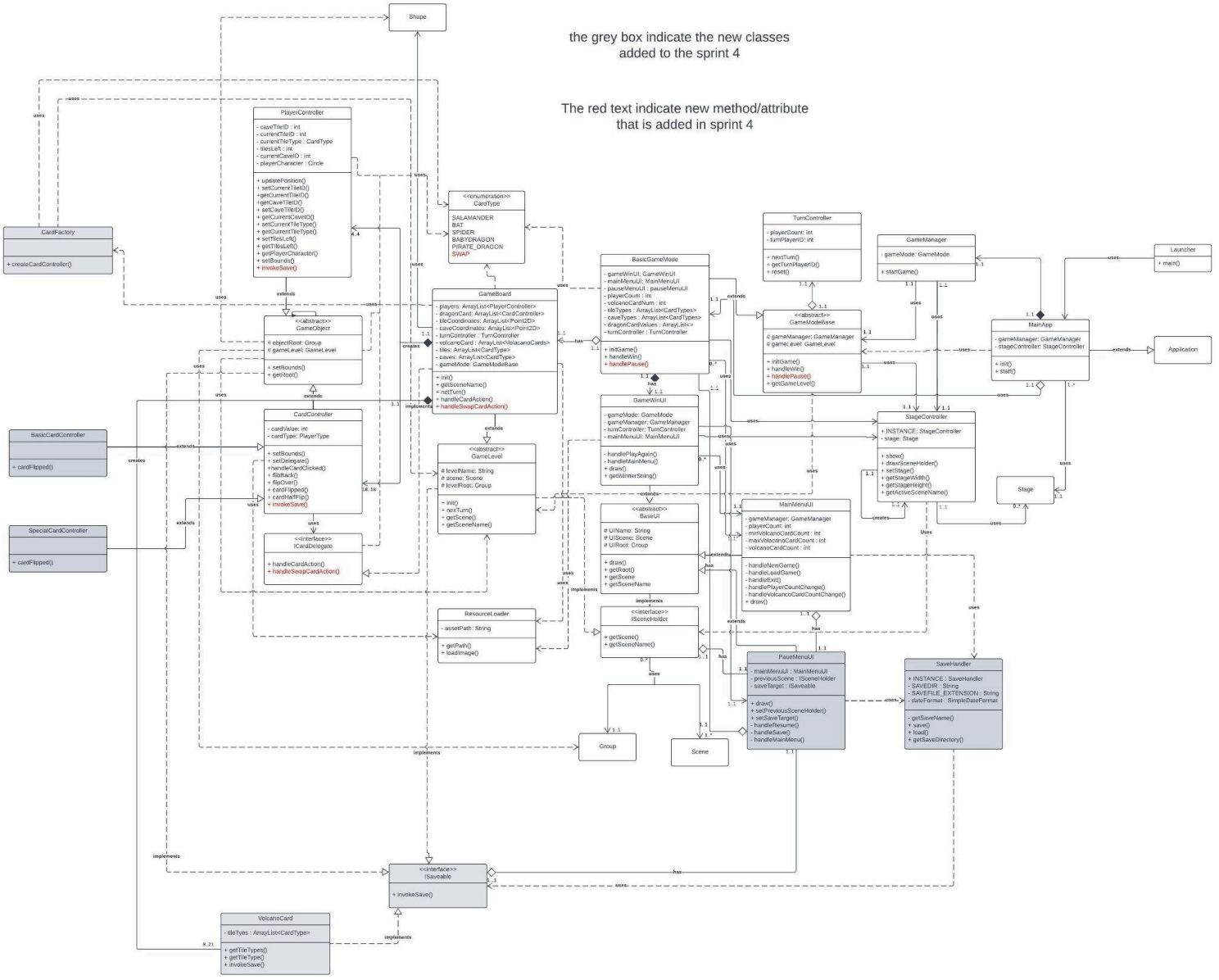
Stimulation
The value of Stimulation emphasises excitement, novelty, and challenge in life. By introducing the mechanic where tokens swap places if a player lands on an occupied tile, we add a layer of unpredictability and excitement to the game. This new behaviour ensures that every move has potential consequences beyond mere movement, making the game more thrilling and dynamic. Players are constantly engaged with the possibility of both positive and negative outcomes, enhancing the overall excitement of the game.

Achievement
Achievement pertains to personal success through demonstrating competence according to social standards. Our swapping mechanic introduces a new skill-based element to the game. Players are rewarded for strategic thinking and precise moves, as they can leverage this mechanic to their advantage by targeting occupied tiles. Successfully executing such a move can hinder an opponent's progress while advancing their own, allowing players to demonstrate their strategic prowess and achieve a sense of accomplishment.

Self-Direction
The value of Self-Direction involves independence in thought and action, fostering creativity, exploration, and choosing one's path. Allowing players to customise the number of volcano cards on the gameboard directly aligns with this value. By giving players the freedom to determine the size of the gameboard, we empower them to shape their gaming experience according to their preferences. This customization option encourages players to explore different game dynamics and strategies, fostering a sense of autonomy and creativity.

Object Oriented Design: UML class diagram (If it’s blurry refer to the page at the very end of this document)

Class Diagram



Reflection on Sprint3 Design
Required Extensions
1st Extension: New Dragon Card 2
In Sprint 4, we successfully implemented the "New Dragon Card 2" extension on our Sprint 3 design, which required a player's token to swap positions with the nearest token on the board, either forwards or backwards, but skips the turn for the current player. Tokens that are already in their caves are not swappable. To achieve this, we introduced a new method, handleSwapCardAction, in the ICardDelegate interface and implemented it in the GameBoard class to handle the token movement logic. We refactored CardController into an abstract class, making cardFlipped an abstract method, and created two concrete subclasses: BasicCardController for the original behavior and SpecialCardController for the new card's behavior, which calls handleSwapCardAction. Additionally, we developed a factory class to instantiate the appropriate card controller based on the card type. This refactoring ensured backward compatibility and facilitated the clean integration of new features, demonstrating effective use of design patterns and scalable code design.

Level of Difficulty: Moderate
The incorporation of this extension was moderately challenging due to the need to add another card controller, refactor CardController into an abstract class, and create a factory to instantiate the correct controller based on the card type. These changes required a careful reorganization of our existing code.

Aspects of Sprint 3 Design Affecting Difficulty:

Distribution of System Intelligence:
The centralized logic for card handling was initially not designed for extension, making it slightly challenging to refactor.

Presence of Code Smells:
The initial tight coupling between CardController and game logic required significant refactoring to decouple and introduce new behaviors.

Use of Design Patterns:
The lack of initial use of the Factory pattern made the introduction of new card behaviors more complex.

Improvements for Future Practice:
If we were to redo Sprint 3 with this extension in mind, we would have initially made CardController an abstract class and used the Factory design pattern for instantiation from the start. This would have allowed for easier integration of new card types and behaviors, reducing the amount of refactoring needed.

2nd Extension: Loading and Saving the Game
For the second required extension of loading and saving the game, it was not hard to design and implement from our Sprint 3 design. The reason behind this is the design leading up to Sprint 3 and subsequently, Sprint 4 had the potential of saving in mind, where we have a singular class that references all objects in the game, from player tokens to the volcano cards, which are all different classes. From this basis, we then utilized interfaces for saveable classes that we can initialize the saving process from a singular point. In terms of code smell, in our opinion, there is none that we know of as we did not utilize reflections, and even if we utilized reflections, it is only for primitive types which is hard to avoid given it is the built-in functionality of Java. The only design pattern worth mentioning is the singleton being used to implement the SaveHandler as we want the saves to be handled at a single point instead of potentially by multiple instances which will lead to IO errors and filename contradictions.

Level of Difficulty: Easy
Designing and implementing this extension was relatively easy due to our initial design considerations for saving and loading. Our architecture had already accounted for centralized control of the game state, which facilitated the addition of saving and loading functionalities.

Aspects of Sprint 3 Design Affecting Difficulty:

Distribution of System Intelligence:
The centralized class managing game objects made it straightforward to implement save/load functionality.

Presence of Code Smells:
Minimal code smells were present, as we avoided complex reflection usage and maintained clear class responsibilities.

Use of Design Patterns:
The Singleton pattern was appropriately used for the SaveHandler, ensuring a single point of control for save operations.

Improvements for Future Practice:
If we were to go back to Sprint 3, we would not change anything in the scope of implementing the saving/loading extension. Our initial design considerations for saving and loading were well-planned, ensuring easy integration of this functionality.

Self-Defined Extensions

1st Extension: New Behavior When a Player Attempts to Move to an Occupied Tile

Incorporating the first self-defined extension, "New behavior when a Player attempts to move to an occupied tile," was relatively easy. This is due to the fact that all we had to do was modify the logic after determining that the target tile is already occupied. This logic is encapsulated in the GameBoard class, making the modification straightforward and localized. This ease of modification speaks volumes about the quality of our sprint3 design.

Level of Difficulty: Easy

Aspects of Sprint 3 Design Affecting Difficulty:

Encapsulation:
By encapsulating the game logic within the GameBoard class, we ensured that any modifications to the game rules or mechanics could be easily managed within a single class. This separation of concerns allowed us to implement the new behavior without needing to make extensive changes to other parts of the system.

Modular Design:
The use of modular components, such as PlayerController and CardController, facilitated the integration of new features. Each component handles specific aspects of the game, allowing us to modify player movement logic without affecting other game elements.

Clear Responsibility Assignment:
Each class in our design has a well-defined role. The GameBoard class handles the overall game state and rules, while individual controllers manage player and card-specific behaviors. This clear separation made it easy to identify where changes needed to be made for the new behavior.

Flexible Data Structures:
The use of flexible data structures, such as ArrayList<Boolean> for tile occupation and ArrayList<PlayerController> for players, allowed us to easily manage and update game state information. This flexibility was crucial in implementing the swapping mechanism.

The incorporation of this extension was straightforward due to the foresight in our initial design. The encapsulation of game logic within the GameBoard class ensured that changes could be localized, while the modular design and clear responsibility assignment facilitated easy integration of new behaviors.

Improvements for Future Practice:
If we were to go back to Sprint 3, we wouldn't really change anything, we would continue to ensure that all game mechanics and rules are encapsulated within specific classes to allow for easy modification and extension. As this approach would further enhance the maintainability and scalability of our codebase.

2nd Extension: Configurable Number of Volcano Cards
For the second self-defined extension of allowing players to choose the number of volcano cards on the game board, incorporating this feature was relatively easy due to the design decisions made during Sprint 3. The key aspects that made this extension easy to address were the clear separation of concerns and the use of variables to store configurable values.

Level of Difficulty: Easy

Aspects of Sprint 3 Design Affecting Difficulty:

Separation of Concerns:
The GameBoard class did not require any changes to accommodate this feature because the number of volcano cards is not fixed in the GameBoard implementation. The board size is determined by the number of volcano cards passed to it during initialization.

Use of Configurable Values:
Since the GameBoard class was already designed to handle a variable number of volcano cards, no modifications were needed to support this extension.

Modifications in BasicGameMode:
The main changes required were in the BasicGameMode class. We updated the BasicGameMode constructor to accept a volcanoCardNum parameter and initialized the volcanoCardNum field. Additionally, the initGame method needed to pass the volcanoCardNum value to the GameBoard constructor.

Separation of UI and Game Logic:
The separation of concerns between the UI and game logic made it straightforward to add this extension. The MainMenuUI class handles the user input for selecting the number of volcano cards, and the BasicGameMode class is responsible for initializing the game with the selected number of volcano cards.

The extension was easy to implement because the design decisions made in Sprint 3 anticipated the need for configurable game parameters. The separation of concerns and the use of variables to store configurable values allowed for minimal changes to the existing codebase.

Improvements for Future Practice:
If we were to go back to Sprint 3, we wouldn’t really change anything. We would continue to emphasize the importance of designing for flexibility and configurability from the outset. Ensuring that game parameters are easily adjustable and that the separation of concerns is maintained will facilitate the integration of future extensions.

Improvement Strategies:

Reflecting on the implementation of all the extensions, we recognize several strengths in our approach, such as the modular design and clear separation of concerns. These aspects allowed for efficient implementation of the new features and demonstrated the effectiveness of our original design decisions. However, there are also areas where we can improve to enhance our future practice:

Strengths:

1. **Modular Design:**
    - The use of modular components facilitated the integration of new features without disrupting existing functionality. Each module handled specific aspects of the game, allowing for easy updates and extensions.
2. **Separation of Concerns:**
    - By separating the UI from the game logic and clearly defining the responsibilities of each class, we ensured that changes could be localized, making the codebase easier to manage and extend.

Areas for Improvement:

1. **Testing and Validation:**
    - **Current Practice:** Our current testing practices focused on validating the core functionality and new features after implementation.
    - **Improvement Strategy:** We should develop and execute comprehensive test plans that include unit tests, integration tests, and user acceptance tests. Automated testing tools can help in running these tests frequently to catch issues early. Implementing a continuous integration (CI) pipeline can ensure that every change is tested before being merged into the main codebase.
2. **Code Documentation:**
    - **Current Practice:** While our code is well-structured, the documentation is minimal and may not sufficiently explain the design decisions or the purpose of complex code blocks.
    - **Improvement Strategy:** Enhancing our documentation practices will make the codebase more accessible to future developers. This includes writing detailed comments within the code, creating comprehensive README files, and maintaining design documentation that outlines the architecture and design decisions. Using documentation tools like Javadoc for Java can also help in generating consistent and readable documentation.
3. **Code Review and Collaboration:**
    - **Current Practice:** Our team collaborated well but could benefit from more frequent and structured code review sessions.

    - **Improvement Strategy:** Implementing regular code review sessions where team members review each other's code can lead to higher code quality and knowledge sharing. Using tools like GitHub pull requests can facilitate this process. Pair programming sessions can also enhance collaboration and collective code ownership.
4. **User Feedback Integration:**
    - **Current Practice:** User feedback was considered but not actively integrated into the development process.
    - **Improvement Strategy:** Establishing a feedback loop with users will ensure that our game features align with user needs and preferences. This can be achieved by conducting regular user testing sessions, collecting feedback through surveys or direct user interactions, and prioritizing user feedback in our development backlog.

Strategies and Techniques for Future Practice:

1. **Continuous Learning:**
    - **Description:** Keeping up with the latest technologies and best practices in game development through online resources, courses, and community forums.
    - **Action Plan:** Encourage team members to participate in workshops, webinars, and online courses. Subscribing to industry newsletters and joining game development forums can also provide valuable insights and keep the team updated on emerging trends.
2. **Agile Practices:**
    - **Description:** Embracing agile methodologies such as Scrum or Kanban to enhance project management and adaptability.
    - **Action Plan:** Implementing agile practices such as daily stand-ups, sprint planning, and retrospectives will help the team stay organized and adaptable. Tools like Jira or Trello can assist in managing tasks and tracking progress.
3. **Peer Collaboration:**
    - **Description:** Engaging in regular discussions and code reviews with team members to share knowledge and improve code quality.
    - **Action Plan:** Scheduling regular peer review sessions and encouraging open communication channels within the team. Tools like Slack or Microsoft Teams can facilitate seamless collaboration and knowledge sharing.
4. **Iterative Development:**
    - **Description:** Adopting an iterative approach to development, allowing for incremental improvements based on feedback and testing.
    - **Action Plan:** Breaking down features into smaller, manageable iterations and releasing updates frequently. This allows for continuous

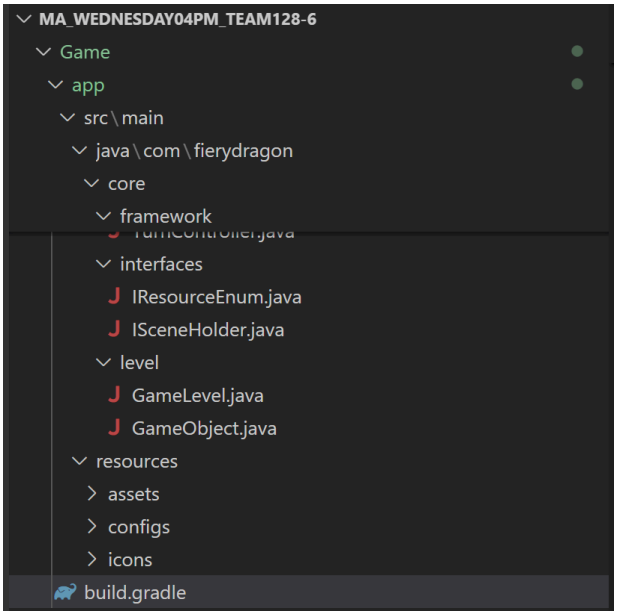
improvement and quick adaptation to user feedback or changing requirements.

5. **Enhanced Testing Practices:**
    - **Description:** Implementing comprehensive testing strategies, including automated testing and CI pipelines.
    - **Action Plan:** Writing test cases for each new feature and integrating automated testing tools like JUnit for Java. Setting up a CI pipeline using tools like Jenkins or GitLab CI to ensure all tests are run with every code change.
6. **Robust Documentation:**
    - **Description:** Creating detailed documentation to support future development and maintenance.
    - **Action Plan:** Using documentation tools like Javadoc to generate API documentation, maintaining an up-to-date architecture overview, and documenting key design decisions and workflows.
7. **User-Centric Design:**
    - **Description:** Focusing on user needs and feedback to drive feature development.
    - **Action Plan:** Conducting user research, usability testing, and feedback sessions to gather insights. Prioritizing features that enhance the user experience based on feedback.

By incorporating these strategies and techniques, we aim to improve our development process, ensuring that our future projects are robust, user-centric, and maintainable.

Instructions on how to build the EXE

IMPORTANT NOTE: Windows Operating system and Java 17 is required to build the EXE, if you have another version of Java instead, navigate to build.gradle



Locate this line of code, change 17 to the version number of Java you have installed on your computer.

```
23  java {
24      toolchain {
25          languageVersion = JavaLanguageVersion.of(17)
26      }
27  }
```

To check the Java version you have installed, you can run this command in the terminal: java -version

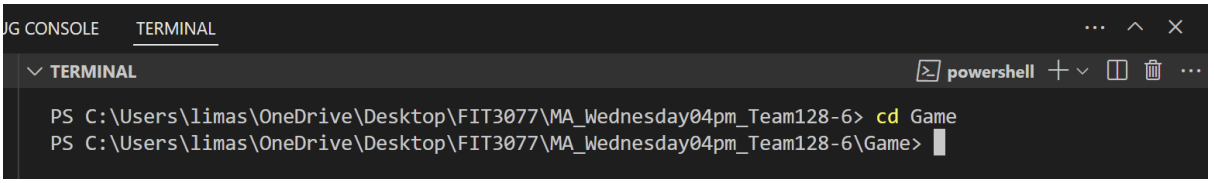
```
PS C:\Users\limas\OneDrive\Desktop\FIT3077\MA_Wednesday04pm_Team128-6> java -version
```

And it will show which version of Java you have installed on your computer.
For example: if you have Java 19 installed on your computer instead of Java 17, change the 17 to 19:

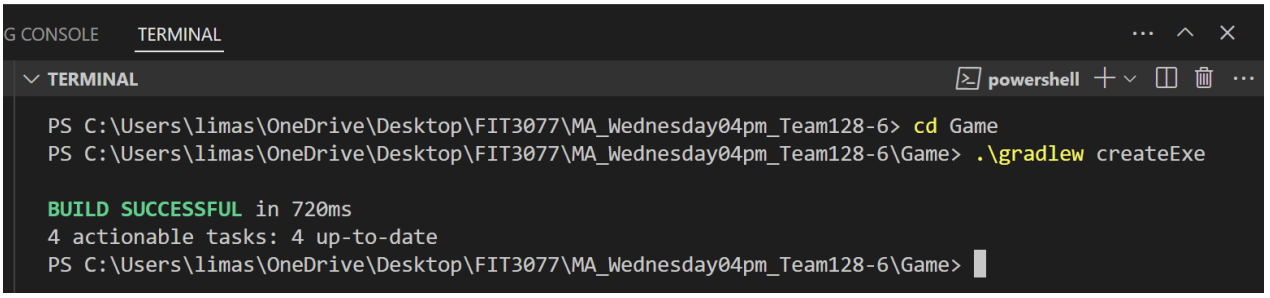
```
23  java {
24      toolchain {
25          languageVersion = JavaLanguageVersion.of(19)
26      }
27  }
```

After that is done, go through the following steps:

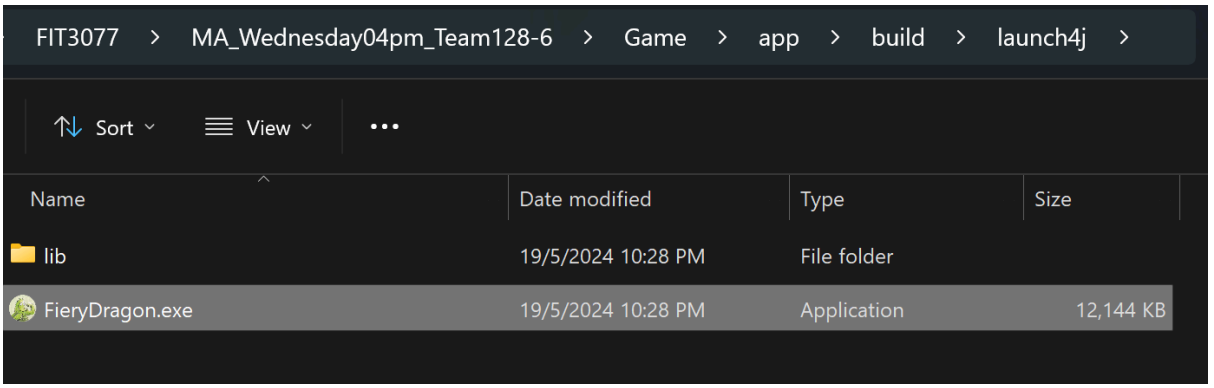
1. In the terminal, navigate to the ‘Game’ folder



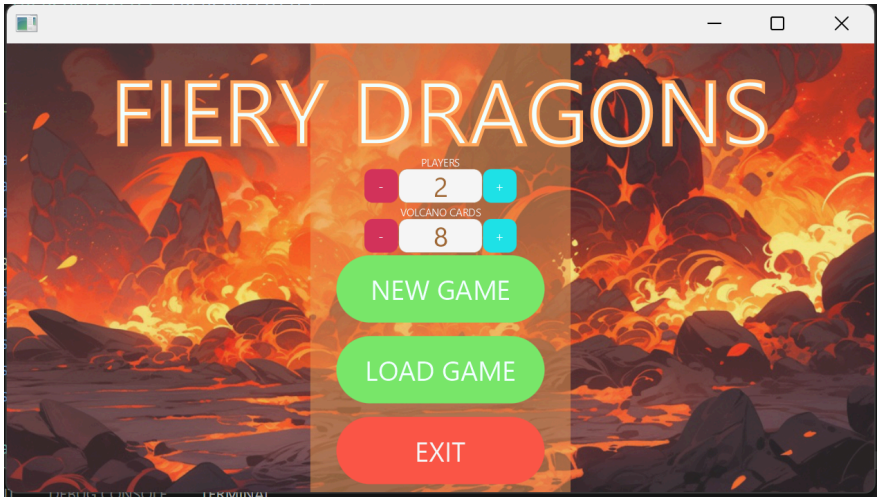
2. Type this command: .\gradlew createExe



3. The EXE will be created, and can be located by going to app\build\launch4j\FieryDragon.exe, note: ‘build’ folder will only appear after steps 1 and 2 have been completed.

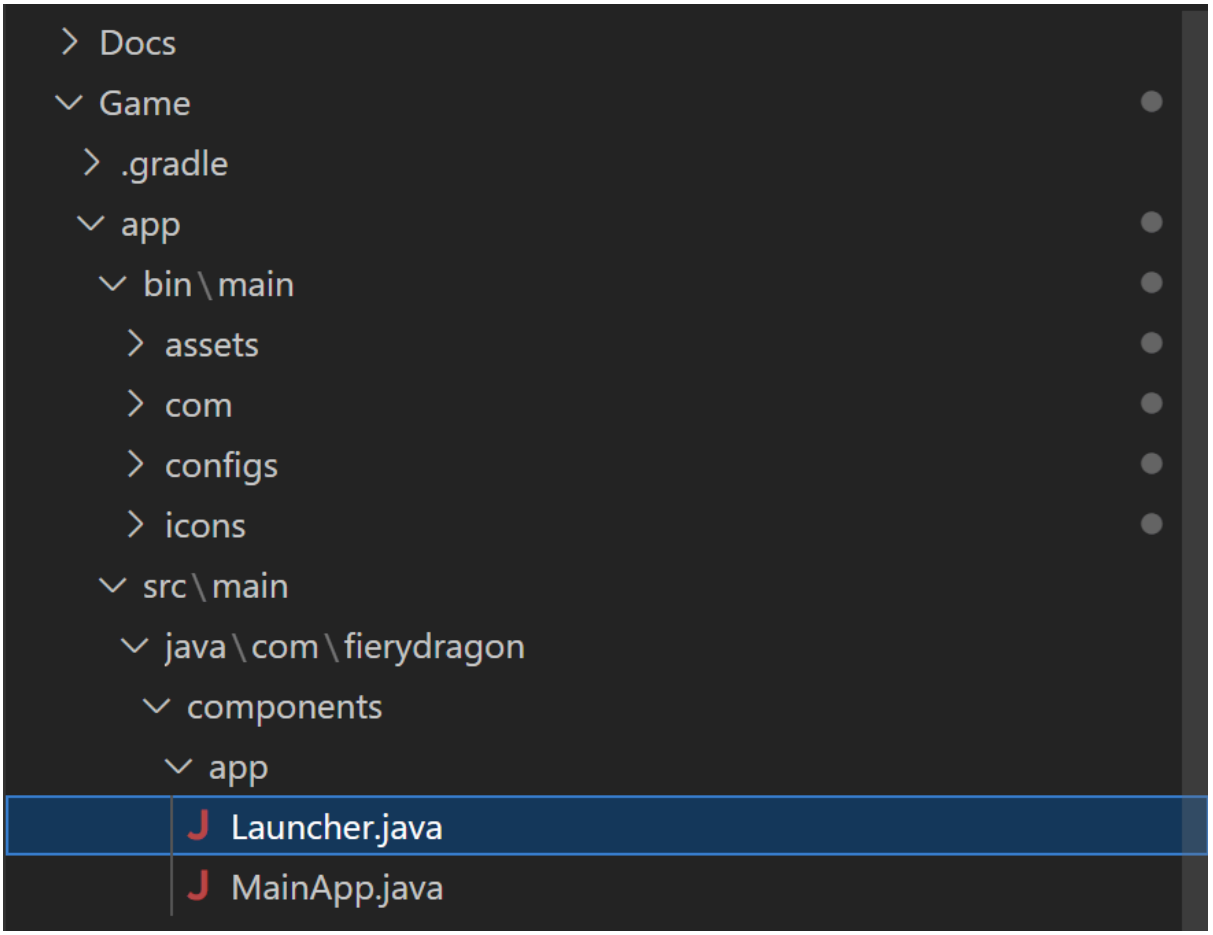


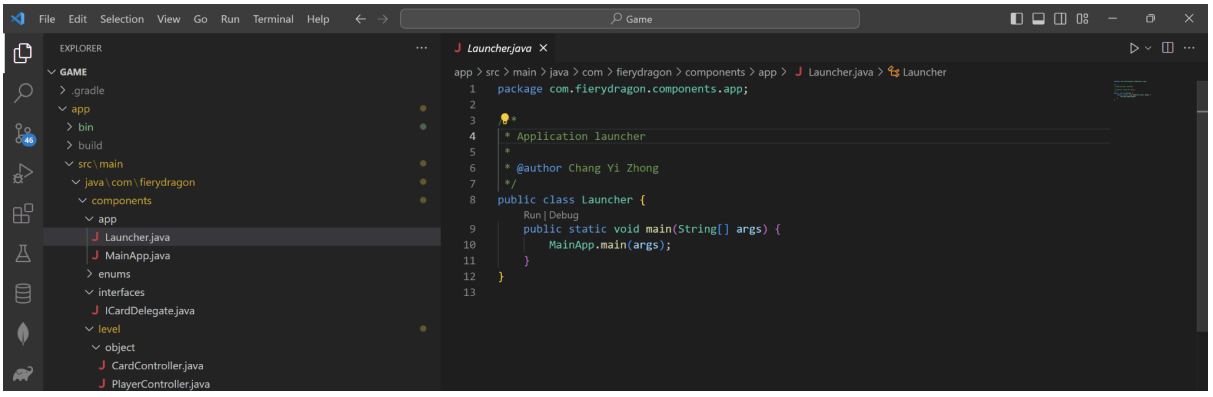
4. To run the EXE, double click on it. The game window will appear, maximize the window to enjoy the full experience



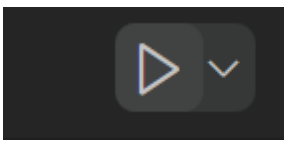
Instruction on how to compile the code:

On VSCode, navigate to app\src\main\java\com\fierydragon\components\app\Launcher.java

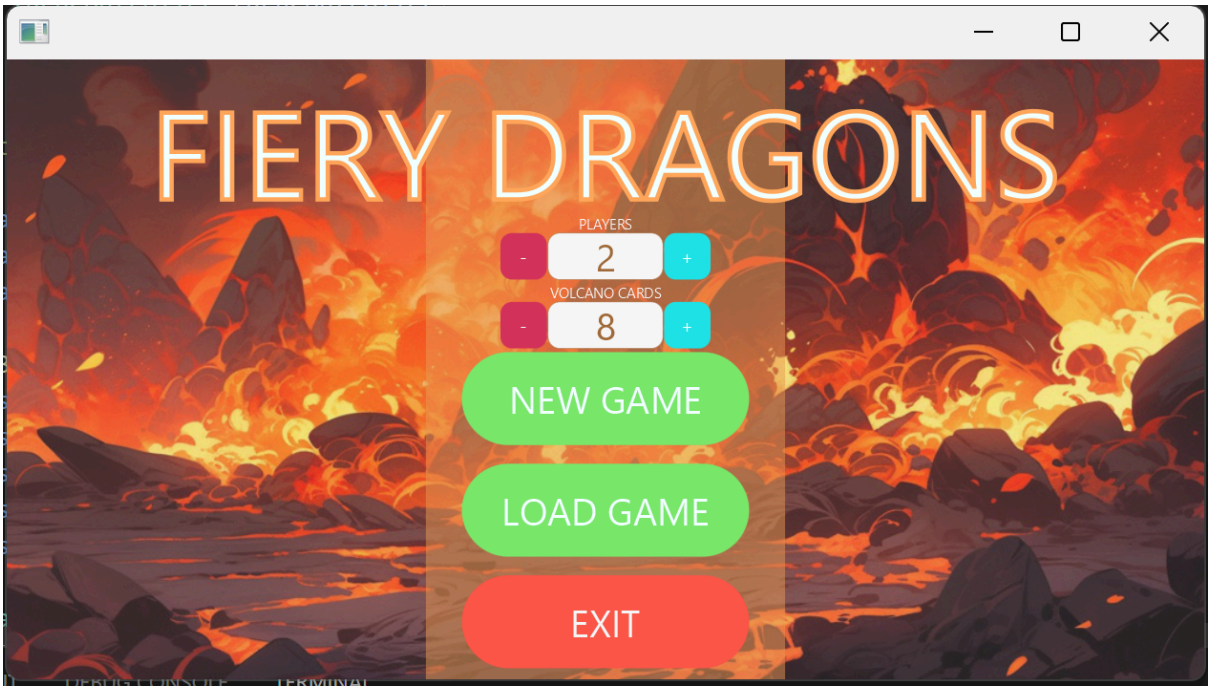




Click the run button on the top right corner:



After that, the game will be launched:



Class Diagram

the grey box indicate the new classes added to the sprint 4

The red text indicate new method/attribute that is added in sprint 4