| FIT3077 System Architecture and Design

SPRINT FOUR DOCUMENTATION

CREATORS:

> Tye Samuels

> Georgia Kanellis

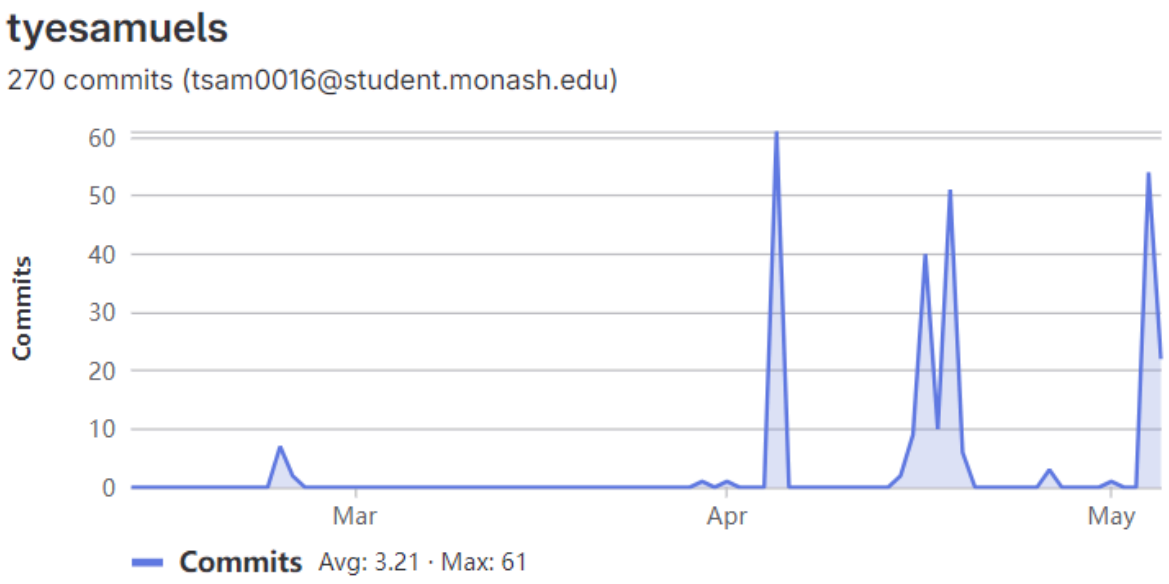
> Audrey Phommasone

> Jun Hao Ng

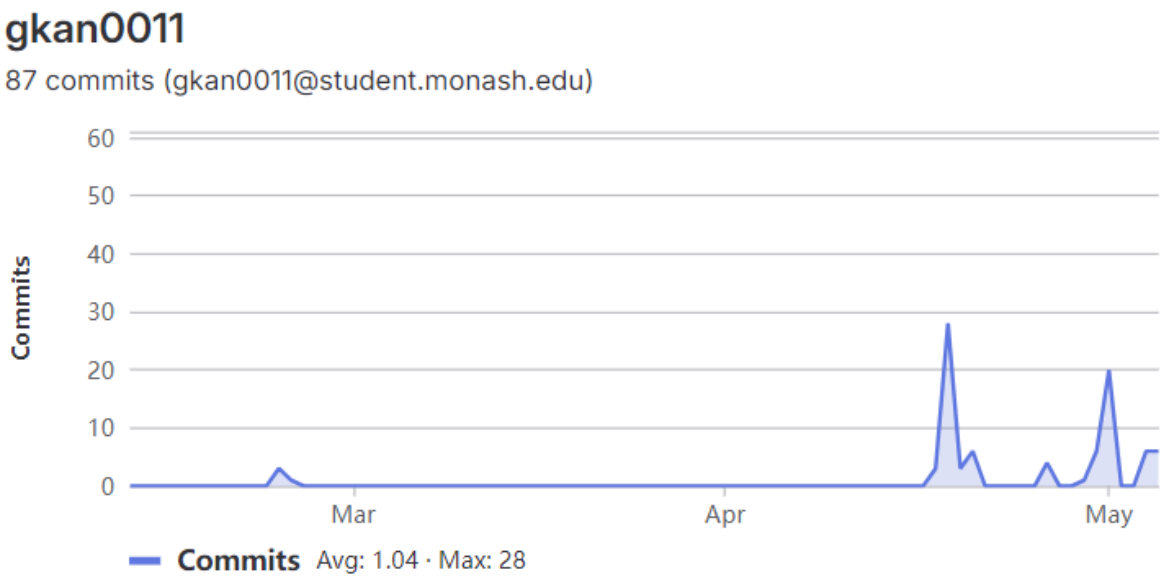
Contents

Contributor Analytics

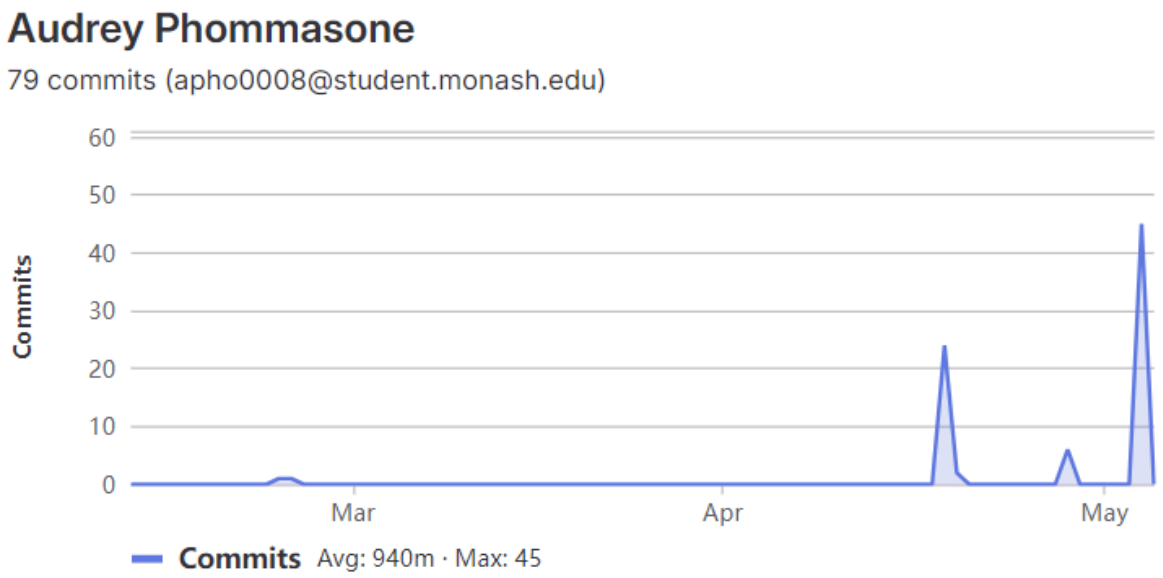
Tye Samuels



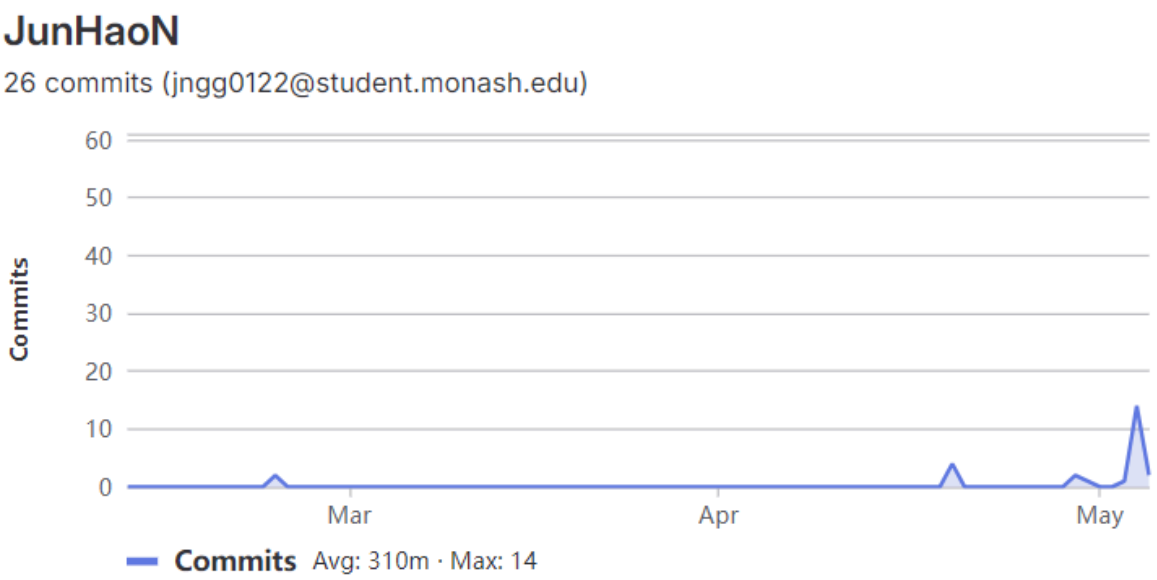
Georgia Kanellis



Audrey Phommasone



Jun Hao Ng



Sprint 3 Reflection

Extensions Analysis

Loading / Saving the Game

This extension is related to the ability for users to save, quit, restart and reload a game to its previous state.

The save and load function for the implementation of Fiery Dragons was a large hurdle to tackle as it involved saving the current game state, and then loading it later, maintaining aspects including the player turn, which chit cards had been turned over, and player positions across the volcano. This was done using the serialisation methods built into Java, making multiple classes implement this interface. The difficult part was understanding all the documentation of Serialisation and its use cases as well as debugging nuanced cases such as static variables being unable to be Serialised. Any class that had an instance of itself saved as an attribute must implement this interface. Finding which classes needed this interface was a mix of trial and error and referencing the class diagram of Sprint 3.

As a result, the process became easier as the whole of the implementation could be saved and serialised for reference of the required attributes to load the game. As Sprint 3 was incredibly well designed and extensible, there is very little of the core functionality to change. No methods were extensively modified in order to incorporate the save and load functionality as Serialisation made the process quite intuitive and only required additional functions and methods to be implemented.

As a result this made FieryDragons more of a God class as it saved an instance of itself in order to access all the variables of the previous game and also loaded the saves in order to start the continuation of the previous game. This also allowed for save files to be singular and not required to piece different files together. Additionally all functionality of extensions and game state were able to be saved consequently.

Rat Rascal

This extension implemented a new dragon card that allowed the token of the player to swap position with the token that was closest to its position, making the player lose their turn afterwards.

This was implemented by making `chit` classes return an `Action` as opposed to a `chitCard` value. Whilst this required a relatively large amount of modifications to the code, this now makes the code more extensible, with added ability for `chitCards` to cause other effects by returning alternative `Actions`. An additional `getNearestPlayer` method was also added in `Player` and in `GameTile` and its child classes. This was implemented by changing the original `setVacancy` method in `GameTile` to `setOccupiedBy` and adding a `getOccupiedBy` method to return which player occupies a space at that time. A `swapPositionAction` was created, using these newly created methods to find the nearest player and be able to swap the locations of these players when executed. Lastly, this was added to the `RascalChitStrategy` which was created to be used by rat rascal chits.

The biggest change that was required to implement this strategy was the switch from `chit` classes returning an integer chit card value, to an `Action`. Whilst the initial implementation was fairly logical given the original requirements, with no particular code smells present, it was not particularly extensible, with the original `getValue` function only being able to serve one purpose: returning an integer of the chit card. Thus, in reflection of Sprint 3, having each `chit` and `chitCard` return an `Action` rather than value would have made this process easier and resulted in a more extensible design overall.

Home Court Advantage

This extension allowed for players selecting chit cards with a matching chit value to their home cave chit to move double the amount of spaces.

This was easy to implement with the given design due to the design's extensibility for many of the classes. Things that needed to be added to the code base included adding an attribute to save the player's cave chit and a corresponding getter method and a method in `chitCard` to pass an external chit for validation to the card's chit. An additional method needed to be created and used to check if the chit matched or not, that differed from the original `validate` method. This was required as the original `validate` method referred to specified `chitStrategy` methods and validated `chits` based on whether they would allow a player to move. This original method was not modified, and instead a `directValidate` method was created. The `getValue` method in `chitCard` needed to be modified to check if the `chit` matched, and double the value if it matched, requiring a parameter of `player` to verify this.

Although many new methods were added for this extension, not much needed to be changed from the original code due to the `chit` and `chitCard` classes adherence to the single responsibility principle.

Thus, in retrospect, no changes would have reasonably been made to Sprint 3 to make this extension easier to implement, as these additional methods were all required specifically for this feature.

Human Value Extensions

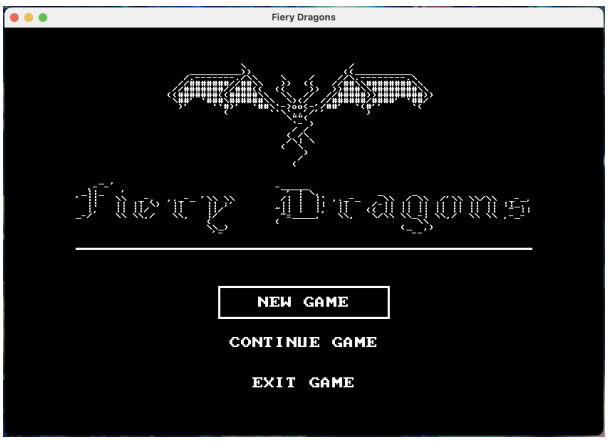
Title / Settings / Load User Interface

Universalism: A World of Beauty

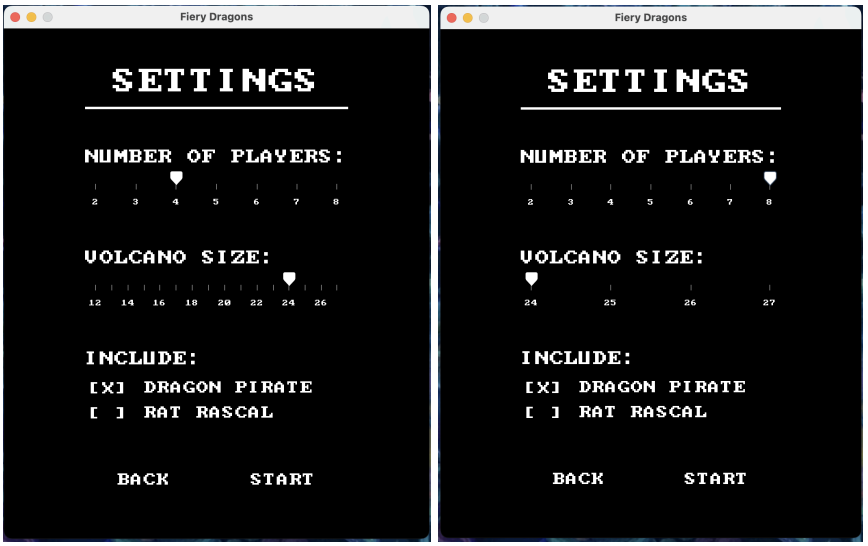
This extension presents a user with a title screen when they first boot up the Fiery Dragons game, as well as a separate setting page. This particular extension addresses the human value of Universalism. The User Interface is pleasant to the eye, and is in-keeping with our ASCII art aesthetic. We designed an interesting ASCII art logo and title text to entice the players.

This was a medium level of difficulty to create as it included creating a `TitleDisplay` class. Though this extension should have been relatively easy to implement, when this extension was initially implemented, the `JFrame` that the game was displayed within was created and held within the `GameDisplay`. This made encapsulating the title screen logic within `TitleDisplay` to be relatively difficult. Transferring the title screen logic to its own class was an awkward task, as there were many unintended dependencies. Once the title screen logic was encapsulated within `TitleDisplay`, the logic was relatively simple.

As stated earlier, the `GameDisplay` originally had far too much system intelligence as it controlled the window, the game board components and the title screen components. This caused the code to “smell”, as the `GameDisplay` abused the object-oriented Single-Responsibility Principle. This required a major refactoring of the whole system initialisation which later became relevant when the `SettingsDisplay` was connected to the game components.



The `SettingsDisplay` was comparatively easier, as it was developed after the methods `displayGameScreen`, `displayTitleScreen` had been created, and thus, the system for displaying and hiding the different page components had been established. Therefore, it was easy to encapsulate the settings components within `SettingsDisplay`.



The `LoadDisplay` was dramatically easier to implement than the previous two screens. This was because the methods for displaying different pages were well established. Thus, implementing this display simply included adding the `JSwing` components to the `JFrame`.

If Sprint 3 could be restarted, we would not contain the title screen logic within the `GameDisplay` class, leading to a major refactoring of system logic. This would have increased the initial readability and maintainability allowing for the later additions of `TitleDisplay`, `SettingsDisplay` and `LoadDisplay` to be introduced more easily.

Changing the Game Settings

Self Direction: Choosing Own Goals / Freedom of Choice

This extension allows the user to customise their game to suit their own wants. This particular extension addresses the human value of Self direction, allowing the player to decide how to approach the game however they choose. The strategy of how the game can be played changes depending on the settings chosen by the user.

This extension was very difficult to implement due to how our components were initialised within Sprint 3, specifically the game board components. This problem occurred because the `VolcanoDisplay`, `PlayerDisplay`, `CardDisplay`, and `InputDisplay` were all initialised within the `DisplayManager` constructor. Thus, these components were all being created at the beginning when the code was first run. These components used hardcoded values like `numPlayers` and others to initialise the game board components, which was unchangeable by the time the users were prompted to select their specific settings.

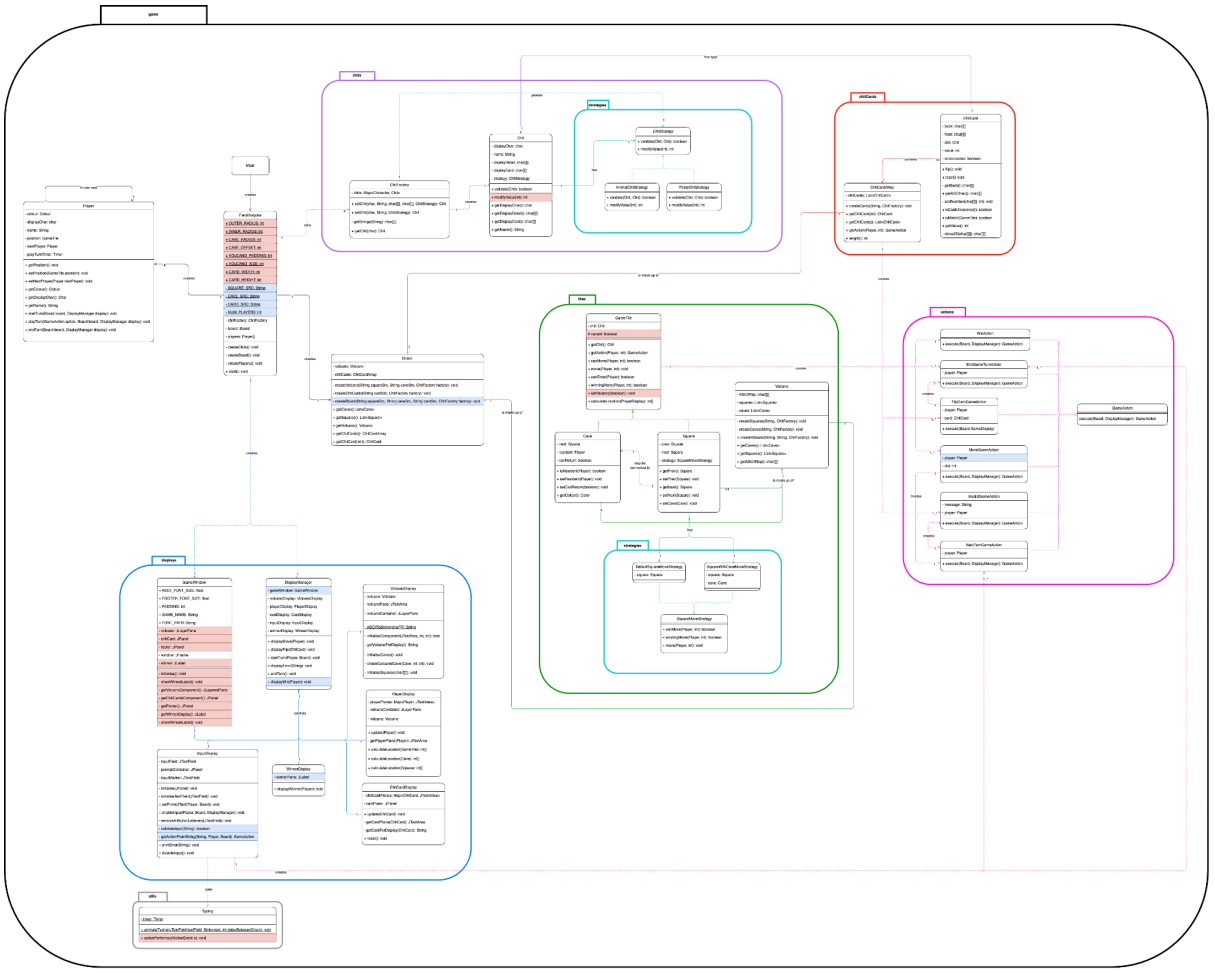
This required some major refactoring as the game board components could not be created before the settings were picked, but simply moving the initialisation to after `pickSettings` caused multiple nullptr errors to occur. This was fixed by separating the components into the method `createGameComponents`. These components were only initialised when strictly necessary, and the system was refactored to only create the game board until after the settings had been selected.

If we were to start Sprint 3 over again, when designing our system, we would consider how our future extensions might work. Thus, we would have realised the issue of initialising the game components before the settings were selected. This would have prompted us to find a different solution when initialising the game components. In hindsight, many of the issues we faced when implementing this extension could have been avoided with proper foresight.

Object-Oriented Design

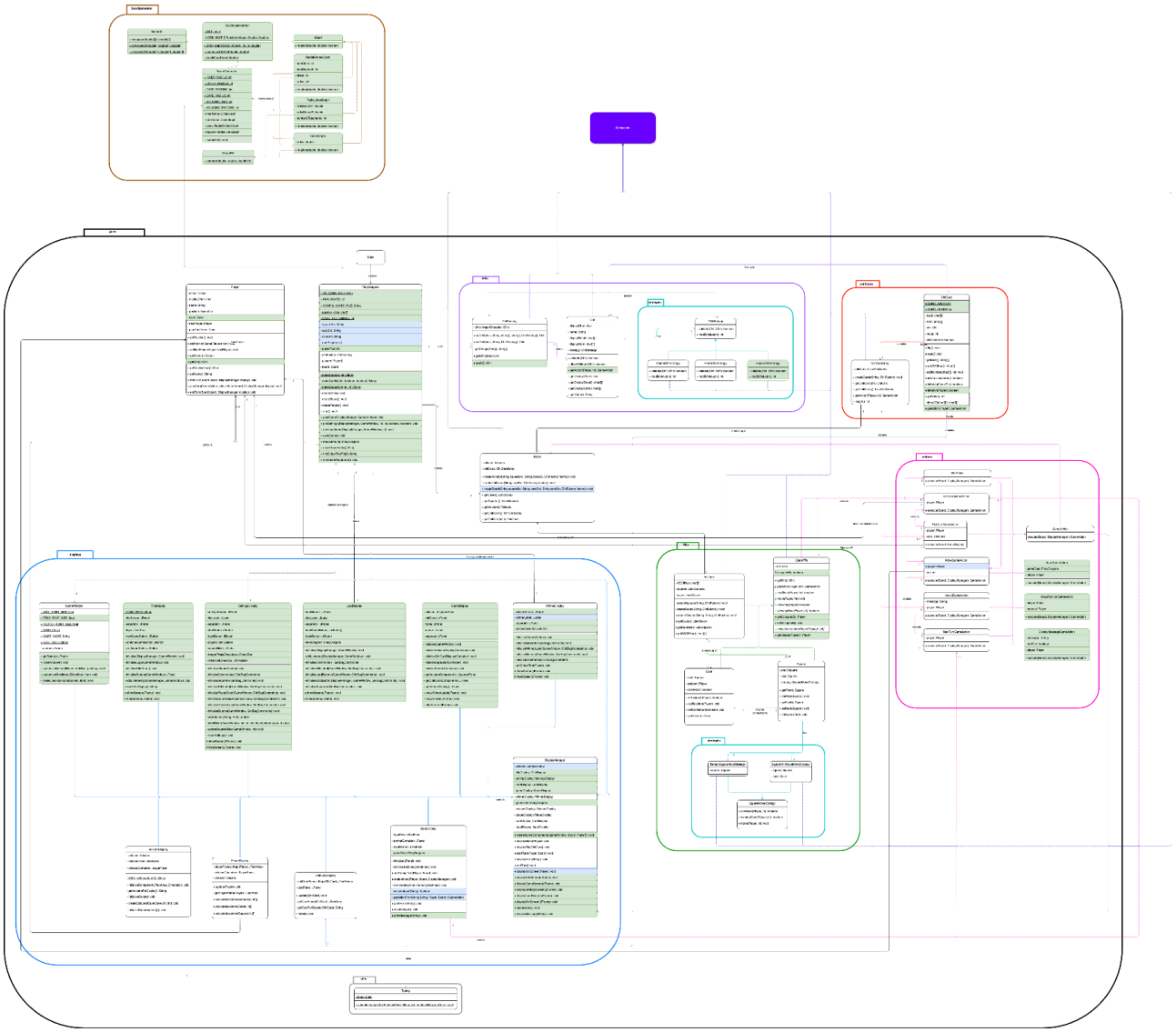
Sprint 3 Class Diagram

See /docs/Sprint 3 Class Diagram.drawio.svg for full image.



Sprint 4 Class Diagram

See /docs/Sprint 4 Class Diagram.drawio.svg for full image.



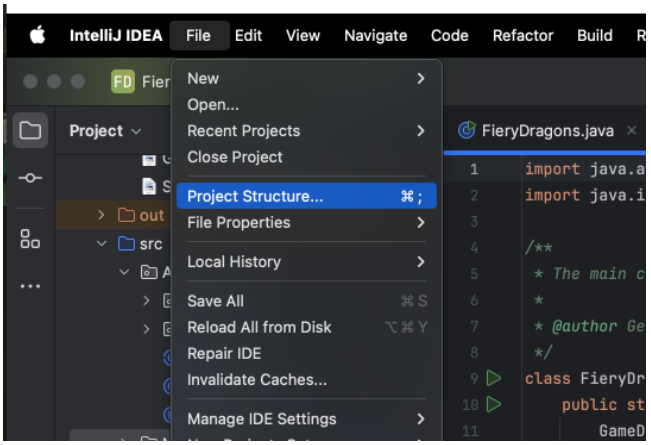
Executable

Executable Description

Opening up the Executable will open a JFrame containing ASCII art displaying the Fiery Dragons title page, with options to open a new game, continue game, or exit game. Selecting the new game button will bring the user to the settings page, allowing them to select a number of players, volcano size, and select extensions for whether the dragon pirate and/or rat rascal chit cards were added to the game. Selecting the continue game button will allow for the user to load a previously saved game and continue the game from where they last left off. Starting a new game or continuing a game will display the Fiery Dragons game, with the volcano on the left, and a spread of Chit Cards on the right. The volcano contains a number of squares, and a cave for each player. The caves are colour coded to match the Player token that currently resides inside them. At the bottom of the screen, the starting player is prompted to pick a chit card using the card index displayed on the screen. By entering a valid number into the game, the player flips the selected chit card. If that chit card has a chit that matches the square that the player is standing on, the player will move along the volcano. The player moves the number of squares that is represented by the value in the corners of the flipped chit cards. By continuously moving along the volcano squares, the player can re-enter their cave. The first player to re-enter their respective cave, will end the game. At this moment, the win screen will be displayed, stating which player won.

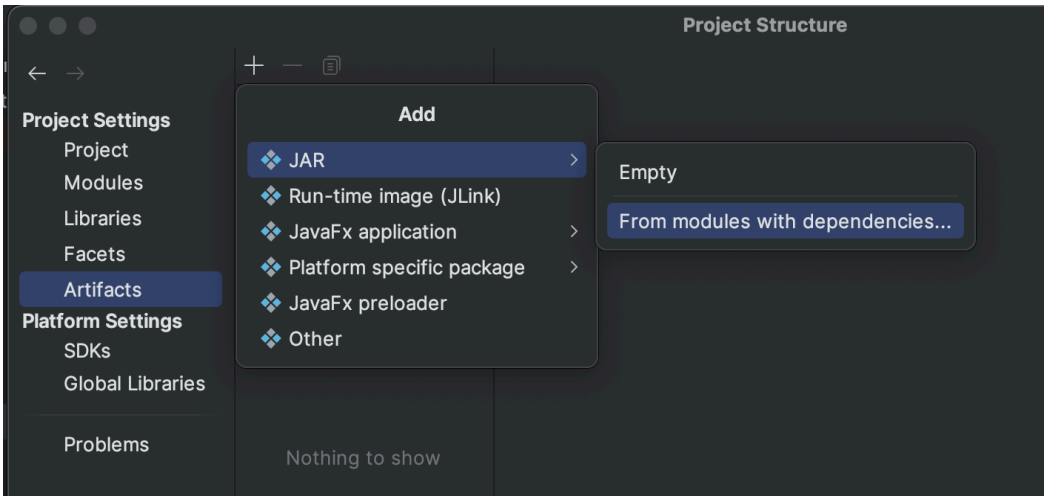
Executable Build Instructions

1. After pulling the source code from either Git or through the zip submission, open up the FieryDragons folder in IntelliJ. Make sure you are running Java 22 on a MacOS or Windows system.
2. Mark the “res” folder as the Resources Root.
3. Open the Main file.
4. Go to the menu bar at the top of the screen and press File to open the drop-down menu. Then press Project Structure...

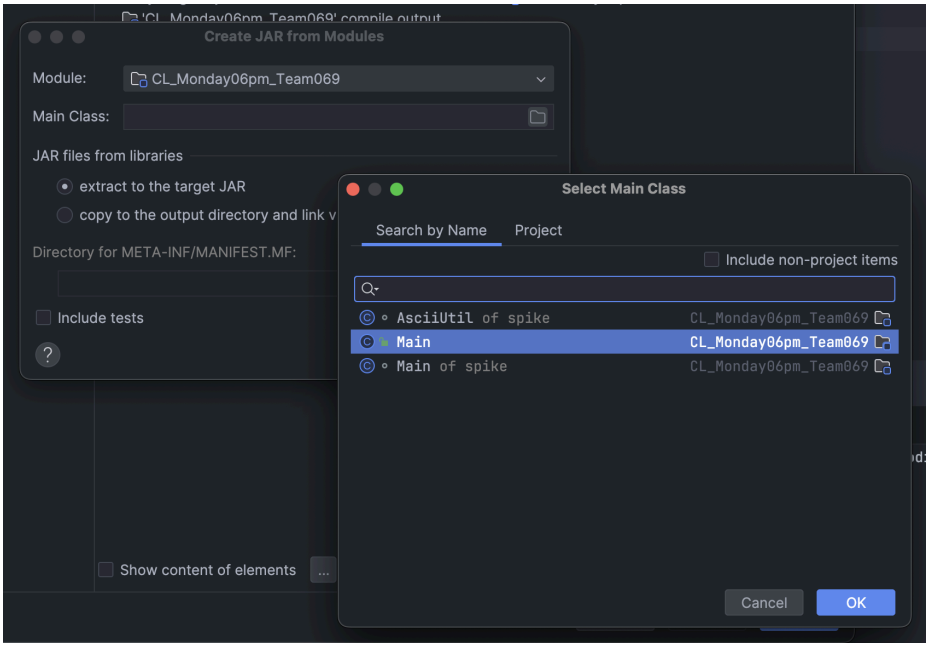


5. A pop-up window should appear. Press Artifacts under Project Settings.

Press the Plus sign, and then add JAR from modules with dependencies.

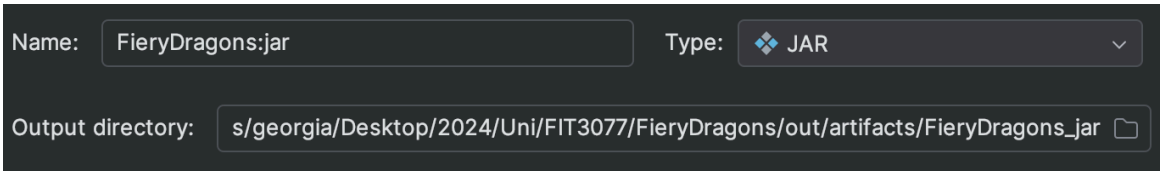


6. Another pop-up window should occur titled “Create JAR from Modules”. Press the file symbol next to Main Class and select Main. Then press OK to bring you back to the “Create JAR from Modules” window

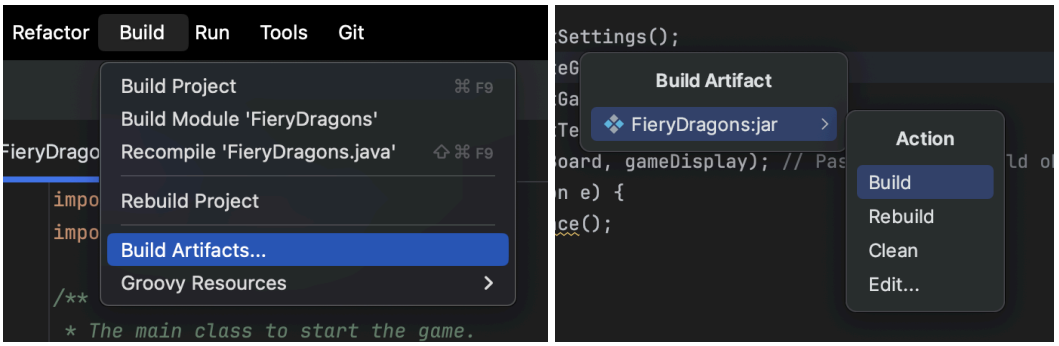


7. Then press the file symbol under “Directory for META-INF/MANIFEST.MF” and select the res folder. Then Press OK.

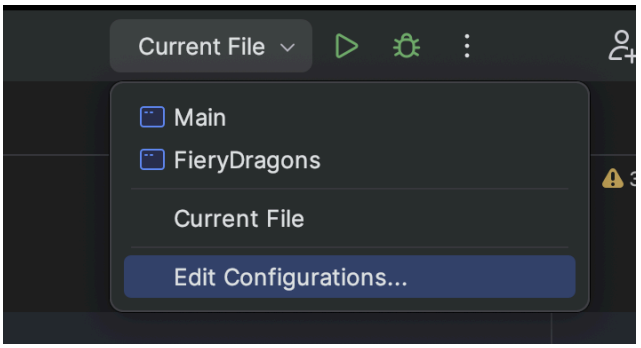
8. Name it “FieryDragons:jar”. Note that the executable will be found in the “folder”.



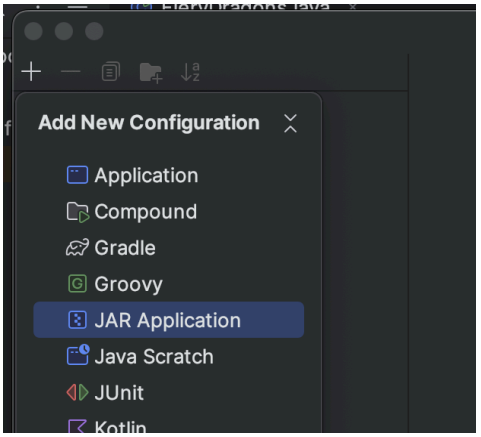
9. Then go to the menu bar at the top of the screen, and select Build to open the drop-down menu. Select Build Artifacts….Then in the new pop-up window, press Build. You should now be able to see the META-INF inside of your res folder.



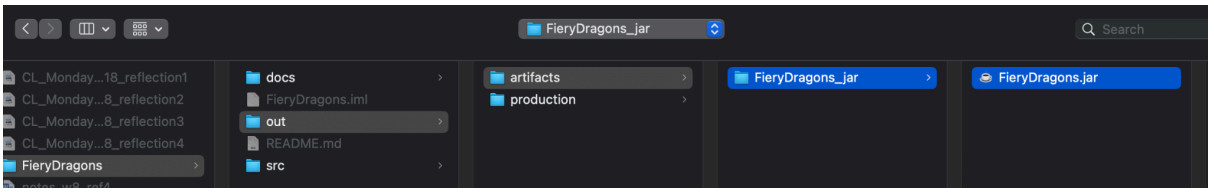
10. Now go to the top right of the IntelliJ window and select Current File, to open the drop-down menu. Then select Edit Configurations…



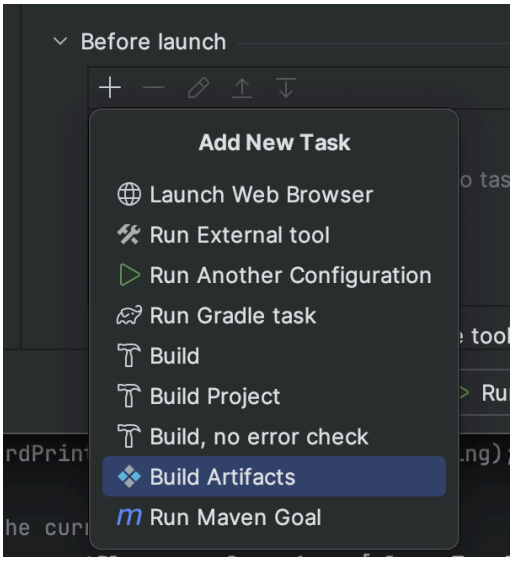
11. A new window should appear named Run/Debug Configurations. Press the plus sign in the top left of the window and select JAR Application.



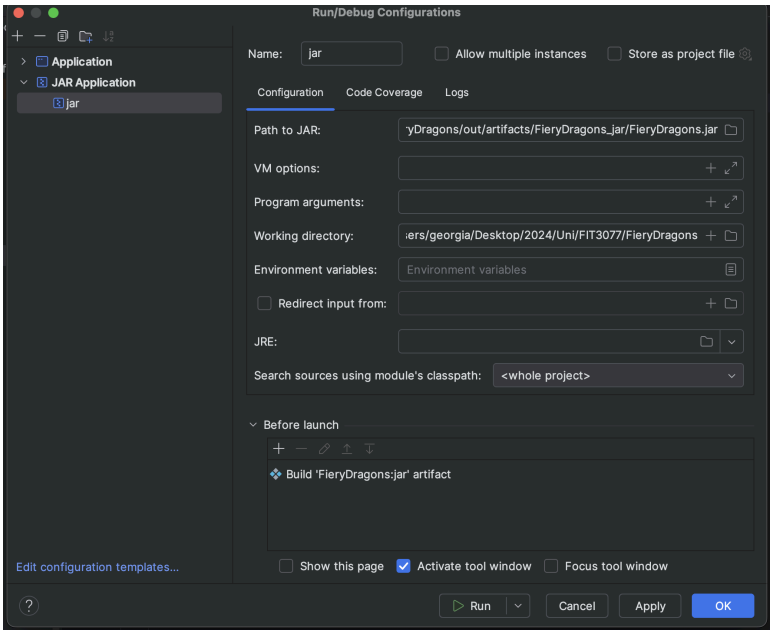
12. Name the JAR Application whatever you like. I have named it “jar” for simplicity. Press the file symbol next to Path to Jar, and select the jar file.



13. Then press the plus symbol underneath “Before Launch” and select Build Artifacts



14. Your screen should look like this.



15. Press Apply, then OK. Now you are capable of running the executable. You have the option to run it through the application within IntelliJ. Or alternatively, you can open the .jar file directly from inside the “out” folder.