FIT3077 System Architecture and Design

SPRINT THREE DOCUMENTATION

CREATORS:

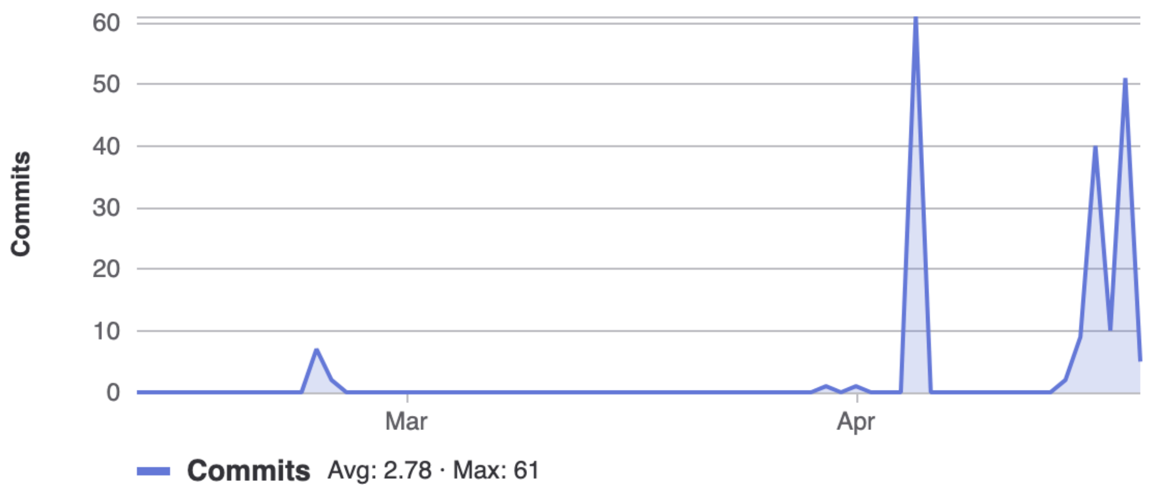
- Tye Samuels
- Georgia Kanellis
- Audrey Phommasone
- Jun Hao Ng

Contents

Contributor Analytics

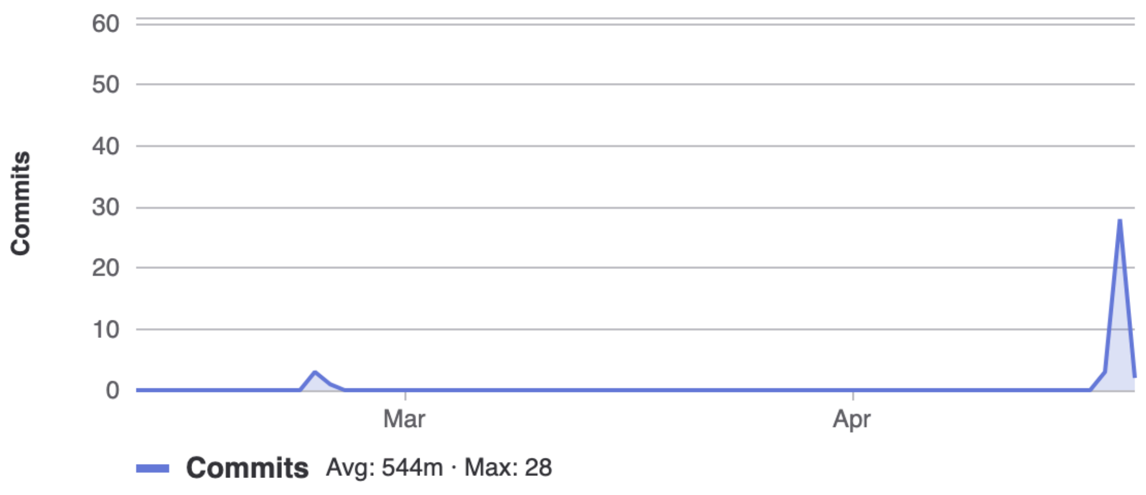
Tye Samuels

189 commits (tsam0016@student.monash.edu)



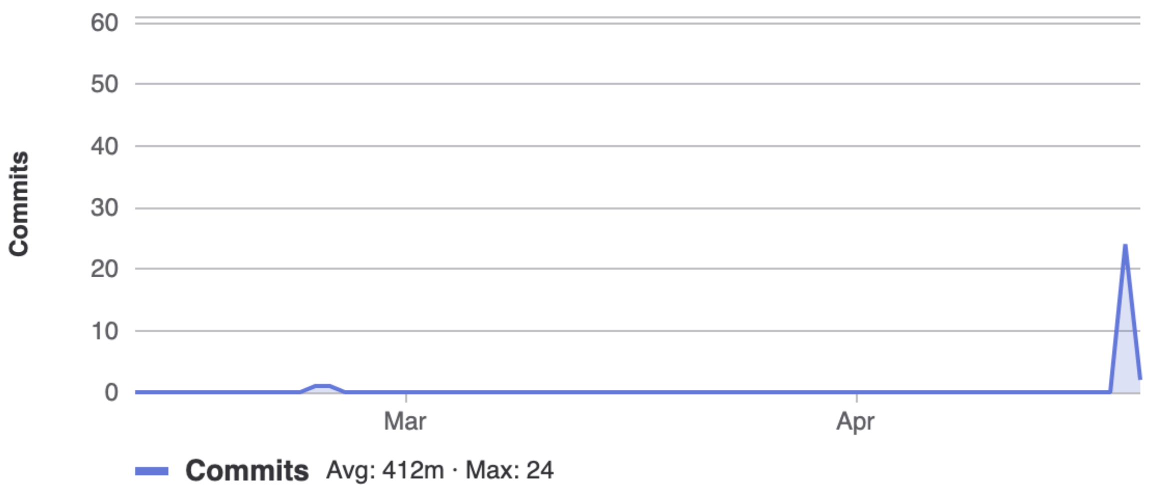
Georgia Kanellis

37 commits (gkan0011@student.monash.edu)



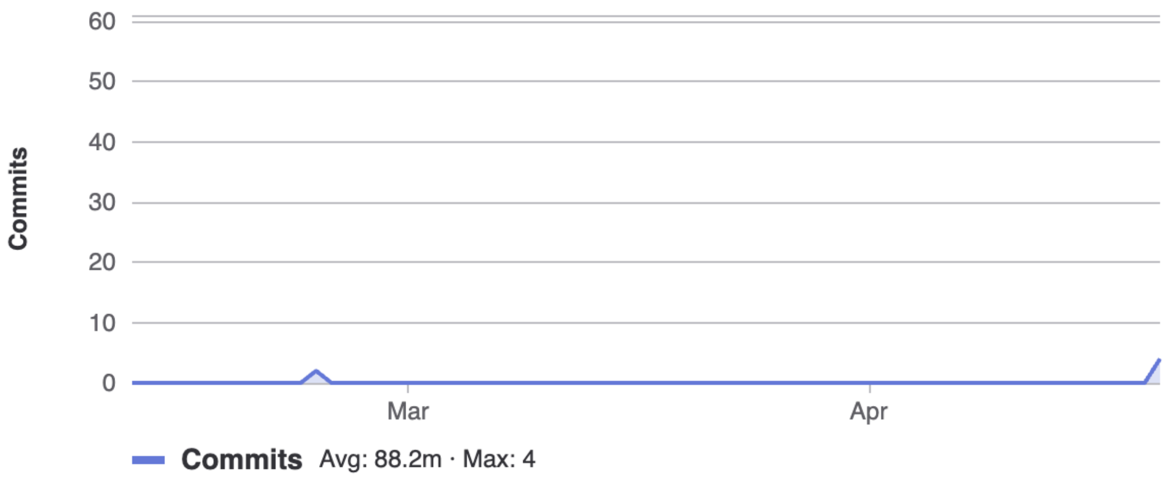
Audrey Phommasone

28 commits (apho0008@student.monash.edu)



Jun Hao Ng

6 commits (jngg0122@student.monash.edu)



Review of Sprint 2 Tech-based Software Prototypes

Assessment criteria

Quality Characteristics

Suitability:

Functional completeness, functional correctness and functional appropriateness

This refers to the functional completeness, correctness and appropriateness of the prototype. An "excellent" prototype is defined to be one that is fully functional, including all required features of the specified tasks and intended user objectives, with full correctness, displaying all features to the display correctly in a way that reflects the internal representation of the game, providing accurate results when used by users. The system should be able to facilitate the specified tasks assigned to the developer's prototype, as well as setting up the initial game board and other key functionalities.

Reliability:

This refers to the system's faultlessness, fault tolerance and recoverability as outlined in the ISO/IEC 20510 quality model. An "excellent" prototype is defined to be one that performs the prototype's specified functions without fault, can tolerate misuse and handle incorrect user input without the system failing.

Interaction Capability:

User engagement

This relates to the aesthetics and user engagement of the prototype. An "excellent" prototype must be easy for the user to understand how to play the game using the information provided by the UI. It must have clear and apparent signifiers to differentiate the tokens, prompts, cards and the board and indicate how a user can interact with the game.

OOP Design Principles:

The following criteria were considered separately to accurately reflect the criteria outlined in the assignment brief for this project. They relate to the quality characteristics of modifiability and maintainability as outlined in the ISO/IEC 20510 quality model.

Single Responsibility Principle:

This SOLID programming principle relates to the concept that objects in the software should manage one specific function only. An “excellent” prototype should have all classes adhering to this principle.

Open-Closed Principle:

This SOLID programming principle refers to the ability for each class to be open for extension but closed for modification. An “excellent” prototype adheres to this principle, allowing for future additions without the need for changing any existing code.

Liskov-Substitution Principle:

This OOP principle refers to the property of superclasses being able to be replaced with objects of their subclass without affecting the correctness of the implementation. An “excellent” prototype adheres to this principle by allowing all parent classes to be replaced by any of their child classes.

Interface segregation:

This OOP principle describes how clients should not be forced to implement or depend upon interfaces or methods it does not need. Thus, an “excellent” prototype has no clients that implement methods they do not require.

Dependency Inversion:

The Dependency Inversion principle indicates that high-level modules should not rely on low-level modules, and instead should rely on abstractions in the interest of promoting decoupling, reducing dependencies. Therefore, an “excellent” prototype should not have high-level classes that rely on low-level modules.

Design Patterns:

Use of Design Patterns and appropriateness recognizability.

Given the project's focus on the use of design patterns in software architecture, there was a higher value for the use of a large number of design patterns where appropriate. In reflection of this, prototypes that did well for this criteria implemented design patterns in all possible areas, with no room for additional design patterns to be implemented to improve the code extensibility.

Individual Sprint 2 Review

Audrey Phommasone → Change of Turn

Suitability: Average 3/5

The implementation for the gameboard allowed for a functional display, correctly displaying distinct player tokens in their locations and illustrating the board game's caves, and volcano cards. The dragon "chit" cards were arranged to the side of the board display for clarity, with each card shown as being turned over, indistinguishable apart from their card numbers which are present for users to be able to select a specific card during their turn.

The change of turn feature was implemented mostly correctly, with players being able to choose whether they continue their turn or skip it to allow the following player to have their turn early. However, there is a bug where flipping a card, whether it was a Pirate or normal Chit card, did not result in an expected result, where players would sometimes skip a turn prematurely or unexpectedly allow a player to continue, regardless of whether the "chit" matched the player's location or not. The feature of the game progressing to the next player when all cards have been uncovered was implemented correctly.

Reliability: Good 4/5

The reliability of this code was sufficient, with all incorrect inputs from users being ignored, and thus preventing the program from crashing due to user input or misuse. However, no error message is displayed to the user, meaning that users will not be informed of why their inputs do not result in any action displayed for the program.

Interaction capability: Excellent 5/5

All information for the game is displayed in a clear and appealing manner, with visually appealing aesthetics. It is relatively easy for a user to understand how to play the game with a prompt and player status displayed above the text field, but would be even more clear with the use of colour and/or increasing the font size for readability and clarity as with other implementation options. Signifiers to differentiate between tokens, prompts, cards and the board were present and worked well, but were even better in other implementations, particularly with the addition of colour.

Single Responsibility: Average 3/5

Most classes were well separated into their individual functionalities, resulting in each class only having one job and thus one reason to change. However, the FieryDragons class was created as a god class, violating this principle. A number of action classes could have been used to improve this further, as implemented by other prototypes.

Open-Closed: Good 4/5

The Open-Closed principle was adhered to relatively well, with most classes being open for extension but closed for modification. In many places however, this could have been improved with the use of strategies to manage differing functionalities. For example, modifications on individual types of cards would not require any additional changes to the cards controller, however any modifications to any of the Location child classes would require some modification elsewhere due to tight coupling in the code. The Open-Closed principle was mostly observed in the implementation of the game board set up, with the board size (number of dragon cards) and number of players (and thus, caves), being adaptable. However, due to the current implementation of the game board as a static text file, these modifications would not be reflected in the display. Therefore, this game board display would be much better implemented with a more dynamic board game display generator, rather than a simple text file.

Liskov-Substitution: Excellent 5/5

The Liskov-Substitution principle was applicable for the Chit Cards and Locations. This was adequately observed by these classes, with the use of parent classes with abstract methods which were implemented by the child classes. No additional public methods were created for any of these child classes, resulting in child classes that could be easily substituted for their parent without issues in the program architecture.

Interface Segregation: Excellent 5/5

No interfaces were used in this implementation, with only inheritance being used for any of the classes.

Dependency Inversion: Good 4/5

Applicable for the Chit Cards and Cards Controller, the Dependency Inversion Principle was observed, with the cards controller managing cards irrespective of their individual types, reducing dependencies. This was also applicable for the Location classes and corresponding child Location classes. Any high level references to Location classes only referenced the abstract Location class, and not any child class of location. The Dependency Inversion Principle could have been further considered in the Square class, which has been given a specific attribute for a Cave instance, resulting in a tight coupling between caves and squares, which could have been reduced by changing this attribute to accept any Location class.

Design Patterns: Excellent 5/5

The implementation demonstrates the use of a number of Design Patterns. This included the use of a flyweight to manage the available options for chit cards, with each chit card being created as a singleton to avoid any errors that may occur when attempting to match chits from chit cards with chits on locations. Furthermore, a singleton was used for the gameboard (implemented as the FieryDragons class) but not for the Display class, with the display being passed through each game method that required some display. As seen in other implementations, this may have been better facilitated with the use of action classes to return things to display, reducing dependencies. A facade was used for the Card Controller and Player Iterator classes, which helped the system to adhere to the dependency inversion principle. Furthermore, an observer pattern was used to manage the resetting of all flipped cards at the start of each player's turn, which allowed for an effective strategy to change the state of a number of card objects without creating unnecessary dependencies and tight coupling.

Summary:

This implementation had a resulting total score of 38/45. The gameboard effectively displayed distinct player tokens and game elements, but had a bug affecting the turn change feature. Reliability was good, handling incorrect inputs without crashing, though lacking user error messages. The interaction capability was excellent, with clear, visually appealing displays, but could benefit from enhanced readability features like colour and a larger font size. The single responsibility principle was generally followed, but violated by the FieryDragons, acting as a god class. The open-closed principle was well-observed, though some areas needed better handling to reduce tight coupling. The Liskov substitution and interface segregation principles were excellently adhered to, with the appropriate use of inheritance. Dependency inversion was good but could have been improved within the Square class. Design patterns were excellently implemented, utilising the design patterns flyweight, singleton, facade, and observer effectively.

Total: 38/45

Georgia Kanellis → Flipping Chit Cards

Suitability: Excellent 5/5

The initialisation and set up of the gameboard was all correct and present, appropriately displaying the dragon “chit” cards to the left of the volcano and its tiles. The cave cards were all present and evenly spread around the volcano. Each volcano tile had its own small animal character that was easy to identify through the key at the top of the display. The cave cards each displayed their animal type through the distinct ASCII art, making them easy to identify. The dragon “player” tokens were each displayed within the respective caves within the display. The player tokens were easily differentiated from each other using different colours and numbers, once again made clear through the key at the top of the display.

The flipping chit card functionality was completed correctly. The user is prompted to select a chit card between 1 and 16, referring to the numbers displayed on the generic chit card display. Selecting a card changes the card’s generic display to the uncovered display. This uncovered display matches the chit card’s random animal type. There is no reset functionality present to make all the uncovered cards flip back to their covered states. Although after discussion, the team decided that this functionality was out of scope as it requires the change of turn functionality to be implemented.

Reliability: Excellent 5/5

The reliability of flipping the chit cards was extremely good, and no bugs were found while testing the functionality. Whenever an invalid card was selected, the system caught it and prompted the user with a unique prompt to try again. The accounted cases included: an invalid integer (i.e. out of range), a non-integer and an already selected chit card.

Interaction Capability: Excellent 5/5

The team was very pleased with the UI of this system. The game board and all relevant information was displayed in a very appealing and distinct manner. The text input for flipping the chit cards was also decided to be very good. We particularly liked the aesthetics of the text delay when prompting the user, simulating a typing effect. It was decided by the team that the UI for Sprint 3 will almost entirely be derived from this system.

Single Responsibility: Average 3/5

Most of the implemented classes were separated into their individual functionalities, particularly the Animal classes. Although the gameboard class violates the single responsibility principle, despite only dealing within initialization. It was suggested that a GameBoard Manager class could be created to help separate the initialisation of the different facets of the game board, particularly the initialization of the chit cards. This would help reduce the amount of code within the GameBoard class.

Open-Closed: Good 4/5

Extending Animals would be incredibly easy, as all that would be necessary is creating a new Animal class that extends the abstract Animal class. No existing code would need to be changed for this to be implemented.

In contrast, the Game board is not very extensible as the display is primarily hardcoded into the system. It would be difficult to implement some future extensions like the other core game functionalities, particularly moving the dragon tokens along the game board display. The hardcoded game board display also makes it difficult in the future during Sprint 4 to implement augmented rules, such as a different sized game board and a different number of players. Thus, it would be better to use a more dynamic approach to printing the game board rather than using a predetermined text file in order to improve the extensibility and decrease the need for modification.

Liskov Substitution: Excellent 5/5

Within the system, the child animal classes are treated the same as the parent Abstract Animal Class. This means that the child classes are easily substituted for the parent class within the system.

Interface Segregation: Excellent 5/5

There is only one interface used within the system, which is the traitor interface allowing the use of the betray() implementation. This allows for any future animals to implement the traitor interface, whilst still extending the abstract Animal class. It also prevents the need for any non-traitor animals to have an implementation of the betray() method, as it would be unnecessary.

Dependency Inversion: Average 3/5

The high level code of GameBoard is decoupled from the instantiation of specific animals through the use of Animal Factories, and such does not depend on the low level specific animals. Despite this, the game board does depend on chit cards, volcano tiles and cave tiles, as they are directly instantiated within the GameBoard class. To reduce this, factories could be used, or as stated earlier the use of managers could help to decouple the high level GameBoard from the low level instantiation.

Design Patterns: Average 3/5

There is good use of Singletons within the system, notably within GameDisplay and GameBoard. It has been agreed that the team should use a Singleton pattern for GameBoard within the Sprint 3 implementation, but not within GameDisplay. The Factory method pattern was also well used, but could be improved using a Flyweight Pattern instead. A Facade pattern would also be useful when creating the chit cards to reduce the amount of code within the GameBoard, GameBoardPrinter and GameDisplay.

Summary:

This implementation had a total score of 38/45. The initial setup of the game board was excellent, with clear and well-displayed elements, and the chit card flipping functionality working flawlessly. Reliability was also rated as excellent, as no bugs were found and error handling was robust. The UI was considered to be excellent, with visually appealing aesthetics and good interaction capabilities. However, the implementation's consideration of the Single Responsibility principle was somewhat a concern, with the GameBoard class needing refactoring. Extensibility was good for the Animal classes but was limited for the GameBoard due to the hard coded elements. The system adhered well to the Liskov Substitution and Interface Segregation principles, but needed improvement to follow the Dependency Inversion principle. Design patterns were used effectively, but there was room for improvement with potential use of Flyweight and Facade patterns.

Total: 38/45

Tye Samuels → Move Dragon Tokens

Suitability: Poor 2/5

The board set up for this implementation was mostly correct, with all elements created and connected correctly. The chit cards are not displayed but are created and randomised correctly.

This implementation was partially complete. MoveAction executes the movement around the board while the GameTile instances validate and perform the actual movement. Instances of this MoveAction are also created, but are not executed by the player. Visual updates to moving tokens are also not implemented. The logic of moving within GameTiles has been implemented, accounting for special cases such as a Square connected to a Cave. Players currently are not able to have turns.

Reliability: Excellent 5/5

The system does not allow for any user input, and thus has no room for system failure due to misuse or incorrect user inputs.

Interaction Capability: Average 3/5

The aesthetics of the system allow for a very aesthetically pleasing UI. However, while the board is displayed clearly, with good signifiers to illustrate the volcano square chit values, and cave chit values in the same way as other implementations; the remaining information for the game is currently missing.

Single Responsibility: Excellent 5/5

This implementation adhered to the single responsibility extremely well, with the display manager class delegating responsibilities as to avoid making a god class. The only place where this principle may have been violated would be in the move function, where validating a move and executing the move aren't separated, however the remainder of the code satisfies this principle well.

Open-Closed: Good 4/5

This implementation followed the open-closed principle well, with most classes being open for extension but closed for modification. The exception to this was the management of the game board display text, with the display being based on a static text file and input strings. In the current implementation, if any of these input strings are changed, the code would break, meaning that future additions or modifications would require changes to the existing code.

Liskov Substitution: Excellent 5/5

The Liskov-Substitution principle was well adhered to, with not much inheritance used in the implementation. Parent classes of actions, chit strategies, square move strategies and game tiles were well implemented, allowing subclasses to be substituted for these parent classes.

Interface Segregation: Excellent 5/5

Clients do not depend upon interfaces they do not use. Thus, interface segregation was implemented correctly.

Dependency Inversion: Excellent 5/5

The dependency inversion principle was well implemented. Whilst the Volcano class has an array of Squares and an array of Caves, as opposed to a combined array of the parent class GameTiles, this was considered to be unavoidable, due to the need to allocate players to caves specifically, rather than any GameTile. The square class depends on a SquareMoveStrategy but only executes it, with no other information about the strategy used. This was the same with the Chit classes, with dependency injection being used. Otherwise, the Display Manager, Players and other classes only interact with their individual displays, rather than the display manager, allowing everything to be handled independently, reducing dependencies in all relevant areas.

Design Patterns: Excellent 5/5

Good use of Design Patterns where applicable. Use of flyweight pattern for creation of chits. A Bridge pattern is used for chit movement, causing a chit to make the player move backwards or forwards. The Bridge pattern is also used to differentiate a Square connected to a Cave from other Squares, altering the implementation of the move related methods. Double dispatch is used to get the location of GameTiles, as the formula to calculate a location is based on whether it is a Cave or Square. A facade has been used as the DisplayManager to make the display subsystem easier to use. Factory methods create all the Squares, ChitCards and Caves from an input string.

Summary:

This implementation did exceedingly well in terms of extensibility but fell short in some areas for functionality. The suitability was rated as poor due to incomplete features such as player movement execution and visual updates. The interaction capability was average, with good aesthetics but missing some game information. Single responsibility was excellent, except for some minor violations in the move function. Use of the open-closed principle was good, though the game board display management was rigid. Liskov substitution, interface segregation, and dependency inversion principles were all excellently adhered to. Design patterns were well-utilised, including the use of flyweight, bridge, double dispatch, facade, and factory methods, enhancing the system's modularity and maintainability.

Total: 39/50

Jun Hao Ng → Win Game

Suitability: Average 3/5

The initial game board is set up correctly in the appropriate format showing all volcano tiles and the 16 game cards. It has some implementation of the core functionality such as flipping the chit cards but it does not show the previously turned cards. It has no sign of player tokens or turns and the user inputs are case sensitive. It does implement a simple game winning mechanic that does demonstrate the appropriate sequence to display the winning sequence.

Reliability: Average 3/5

The user input does handle inappropriate inputs relatively simply and does not break easily, the rest of the display is editable due to neglect on certain function implementations however the core functionalities are not affected.

Interaction Capability: Good 4/5

The game board and cards are clearly defined and easily visible. The user actions and inputs can be found relatively easily and are easy to operate. The formatting and UI could be vastly improved, particularly in the sizing and use of a wider rectangle for the user input to provide a more clear signifier, but the overall design does suffice.

Single Responsibility: Average 3/5

There is a mixed usage of a god class and single responsibility classes. The GameBoard does several things at once, such as initialising not only the display but also all the dragon cards and game start. However the ASCII and any visual functions is handled by its own object classes

Open-Closed: Average 3/5

In the future for introducing the other key functionalities. ASCII is closed for modification, everything that utilised ASCII will use the interface. No existing ASCII code will be modified. Gameboard will always have to be modified with any additional functionality.

Liskov Substitution: Excellent 5/5

No abstraction or inheritance was used.

Interface Segregation: Excellent 5/5

No classes that do not utilise ASCII art implement the DragonInterface. Otherwise, this is not applicable as there is no abstraction.

Dependency Inversion: Excellent 5/5

No abstraction or inheritance was used.

Design Patterns: Poor 2/5

The cardFactory class attempts to use the Factory method pattern, but is not fully implemented due to outside key functionalities. Otherwise no patterns are implemented.

Summary:

This prototype implemented some core functionality but had room for improvement in the design architecture. The game's initial setup and some key features were present, such as flipping chit cards and a basic winning sequence, but lacked player tokens, turn mechanics, and handled user inputs case-sensitively. The game board and cards were clear and user-friendly, but had some room for improvement in the UI. The prototype partially adheres to single responsibility but also has a number of god classes, with some refactoring needed. It met the open-closed principle in the use of ASCII art, but not for GameBoard. Design pattern usage was poor, with an incomplete Factory method pattern implementation.

Total: 33/45

Sprint 3 Outline

Game Board Setup

A common issue around the visual representation of the Volcano is the fact that it has been hardcoded, thus the game tiles must also be hardcoded to match the chits on each tile. Tye’s code begins to solve this problem by using a string to specify the type and order of chits on the Caves and Squares, in addition to the type and value on ChitCards; however, for the Volcano, the input string must match the hardcoded Volcano ASCII art in order for the game to function correctly, where the Chit displayed on the screen matches the Chit on the corresponding GameTile in the backend. In this sprint, using these input strings, the ASCII representation of chits can be dynamically placed on a premade empty Volcano; this idea will be further extended in Sprint 4 when the empty Volcano will also be dynamically generated.

The creation of the Chits used by the game is simplified through Tye and Audrey’s use of the Flyweight Pattern, with all entities in the game sharing Chit instances, rather than cluttering the game with many different Chit instances, essentially all serving similar purposes. Chits stores all ASCII art associated with the Chit, used by the displays, in addition to a method for validating if another Chit is a match and changing the value returned by flipping a card (e.g. for Pirates, this value is negated so the Player moves backwards). Tye’s use of the ChitStrategy interface makes use of the Bridge Pattern to define behaviours of the Chit at run time, allowing for more flexible code. This separates the Chit class from its implementation, allowing the implementation to change during run time; thus the system doesn’t require a new class to represent the Pirate Chit.

As was done in many of the team’s implementations, a ChitCardArray will hold the ChitCards, implementing Tye’s use of a csv string to simplify ChitCard creation. Georgia’s elegant method of shuffling cards, using the inbuilt Collections.shuffle is the chosen method for Sprint 3.

Tye’s use of a DisplayManager, acting as a facade to simplify how other components of the game interact with the display will be expanded upon in this sprint. Moreover, for Sprint 3 our game UI will follow Georgia’s layout due to its aesthetic superiority;

however, her implementation will be made more extensible by better utilising the Swing library and creating display classes to handle specific sections of the interface.

Change of Turn

When considering how to implement game turns, it was important to prevent the creation of “god classes”. With many of the Sprint 2 implementations, a while loop representing the entire game loop was designated as an indicator of the presence of a “god class”. Tye’s code increases the modularity of a game turn, by taking the responsibility of controlling all player turns from an overarching class, making Player’s handle their own turn and pass control over to the next Player when their turn is complete. The FieryDragons class simply tells the first Player to start their turn, and the chain of game turns begins.

By representing actions performed by the Player (e.g. flip a card or move) with a GameAction class, Tye’s code makes use of the Command Pattern. This method reduces dependencies, the Player is not required to know what classes are involved in an action, they simply execute the action and the GameAction class delegates its execution to the necessary classes.

Flipping Chit Cards

The flipping functionality was implemented by Georgia within Sprint 2, where it was done within the current player directly. Within Sprint 3, the flipping is instead implemented using the FlipCardGameAction. The use of actions was an idea suggested by Tye, as it simplifies the system. The use of actions follows the Command Design pattern, helping us reduce dependencies. This implementation allows the player to not worry about how an action is executed, as it is handled within the action itself.

To create a new FlipCardGameAction, the player must first input a string into the input display and press enter. The input design was taken mainly from Georgia’s Sprint 2 implementation, using similar verification, and aesthetics. Once an input is entered, the getActionFromString is called, which validates the string. If the input was valid, the getAction method is called, which creates a new FlipCardGameAction. By executing a FlipCardGameAction, the selected card’s state is flipped to be uncovered, switching its display to show its uncovered card ASCII design, and checks to see if the chit type matches the chit type at the player's location.

Move Dragon Tokens

By making GameTiles validate and perform moves, the implementation supports a modular approach, with each tile passing a call to move to the next tile until the destination is reached, this eliminates the presence of a "god class" externally searching through GameTile instances by utilising a Chain of Responsibility Pattern. Using the Bridge Pattern, SquareMoveStartegies allow a Square to alter the algorithm for calculating the next move, reflecting the difference between a normal Square and Cave connected Square without requiring a new dedicated class. This also allows the implementation of a Square to change at run time, without requiring an external mechanism to dictate the Square's SquareMoveStartegy.

After flipping a card, if it matches the Chit where the Player is currently standing, getAction is called on the GameTile occupied by the Player. Tye's implementation uses polymorphism to adjust the GameAction returned depending on the context of the movement: if the move is invalid, EndTurnGameAction is returned, if the move is valid MoveGameAction is returned, if the movement would result in the player winning a WinAction is returned. This use of polymorphism ensures the extensibility of the system, supporting the Open-Closed Principle, additional conditions can be placed on the return of a GameAction, with respect to movement, without causing error.

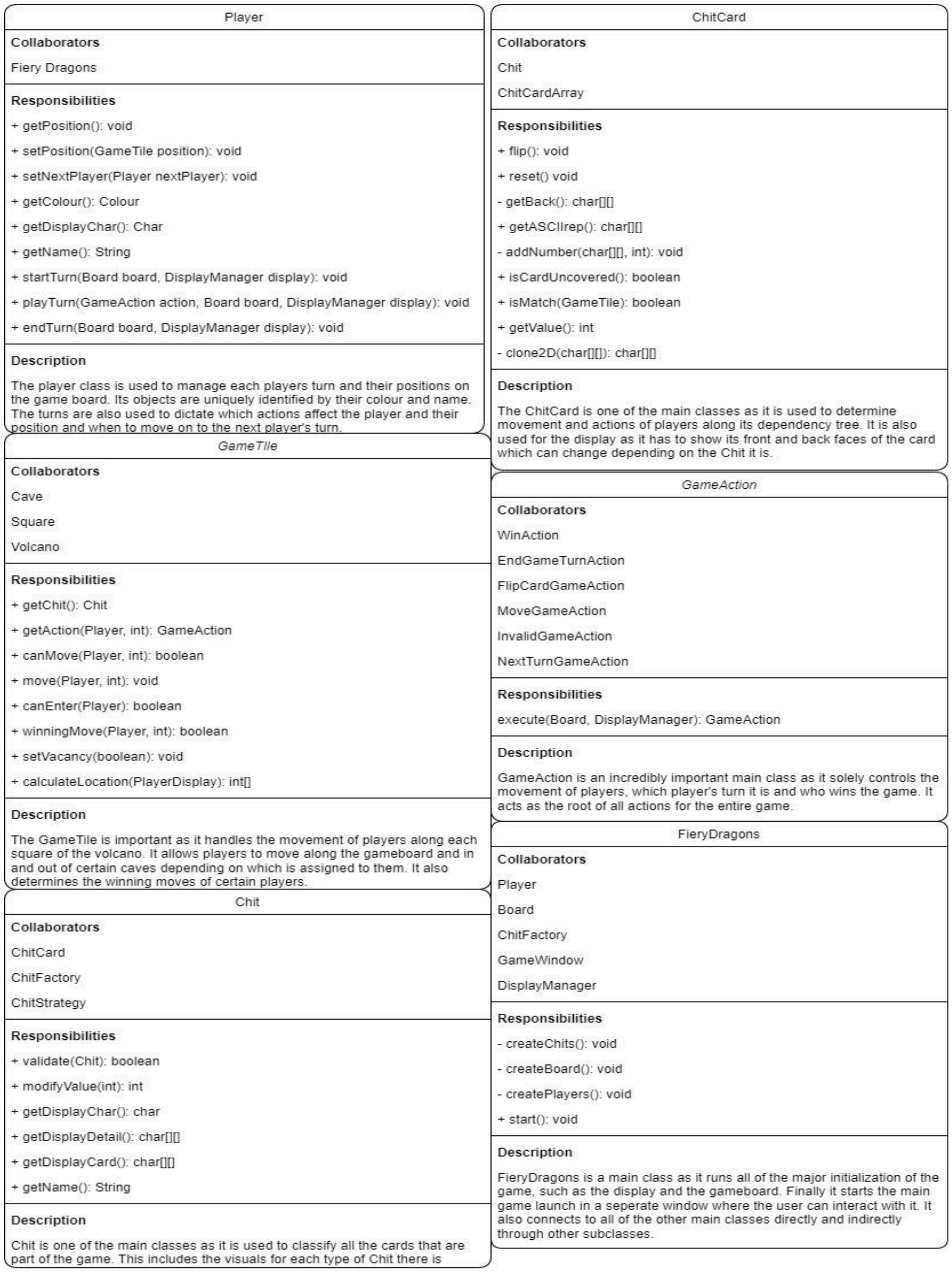
Win Game

The win game functionality in Sprint 2 was implemented in Jun's prototype, with the use of an endGame() function to be solely responsible for handling the display of the game's winner. For this sprint, it was decided that a similar strategy would be used. As this functionality required the implementation of other functionalities, additional ideas will need to be utilised to implement the full win game functionality. These were outlined in Georgia's discussion in Sprint 2, where the game was to be implemented such that the game is complete once a player has traversed the entire length of the game board. This was considered by a canMovePlayer method which compared the player's distance from their home cave to the number displayed on a chit card. The planned implementation of this for Sprint 3 is outlined in the above section "Move

Dragon Tokens”. Georgia’s prototype then deduces a winner based if a player has reached their home cave, calling a winGame function in a similar way to that of Jun’s implementation. Thus the improved implementation will run in a relatively similar manner, but with the addition of Actions as commands (as to be implemented in other parts of the architecture), employing the use of a WinGameAction. Executing this action should call some endGame function implemented in the main display facade (DisplayManager), as well as activate some WinDisplay class in the main display, which will be used to handle the display of specific user information of the winner in the final win screen.

Object-Oriented Design

CRC Cards



Player

Collaborators

Fiery Dragons

Responsibilities

+ getPosition(): void
+ setPosition(GameTile position): void
+ setNextPlayer(Player nextPlayer): void
+ getColour(): Colour
+ getDisplayChar(): Char
+ getName(): String
+ startTurn(Board board, DisplayManager display): void
+ playTurn(GameAction action, Board board, DisplayManager display): void
+ endTurn(Board board, DisplayManager display): void

Description

The player class is used to manage each players turn and their positions on the game board. Its objects are uniquely identified by their colour and name. The turns are also used to dictate which actions affect the player and their position and when to move on to the next player's turn.

GameTIle

Collaborators

Cave

Square

Volcano

Responsibilities

+ getChit(): Chit
+ getAction(Player, int): GameAction
+ canMove(Player, int): boolean
+ move(Player, int): void
+ canEnter(Player): boolean
+ winningMove(Player, int): boolean
+ setVacancy(boolean): void
+ calculateLocation(PlayerDisplay): int[]

Description

The GameTile is important as it handles the movement of players along each square of the volcano. It allows players to move along the gameboard and in and out of certain caves depending on which is assigned to them. It also determines the winning moves of certain players.

Chit

Collaborators

ChitCard

ChitFactory

ChitStrategy

Responsibilities

+ validate(Chit): boolean
+ modifyValue(int): int
+ getDisplayChar(): char
+ getDisplayDetail(): char[][]
+ getDisplayCard(): char[][]
+ getName(): String

Description

Chit is one of the main classes as it is used to classify all the cards that are part of the game. This includes the visuals for each type of Chit there is

ChitCard

Collaborators

Chit

ChitCardArray

Responsibilities

+ flip(): void
+ reset() void
- getBack(): char[][]
+ getASCIIrep(): char[][]
- addNumber(char[][], int): void
+ isCardUncovered(): boolean
+ isMatch(GameTile): boolean
+ getValue(): int
- clone2D(char[][]): char[][]

Description

The ChitCard is one of the main classes as it is used to determine movement and actions of players along its dependency tree. It is also used for the display as it has to show its front and back faces of the card which can change depending on the Chit it is.

GameAction

Collaborators

WinAction

EndGameTurnAction

FlipCardGameAction

MoveGameAction

InvalidGameAction

NextTurnGameAction

Responsibilities

execute(Board, DisplayManager): GameAction

Description

GameAction is an incredibly important main class as it solely controls the movement of players, which player's turn it is and who wins the game. It acts as the root of all actions for the entire game.

FieryDragons

Collaborators

Player

Board

ChitFactory

GameWindow

DisplayManager

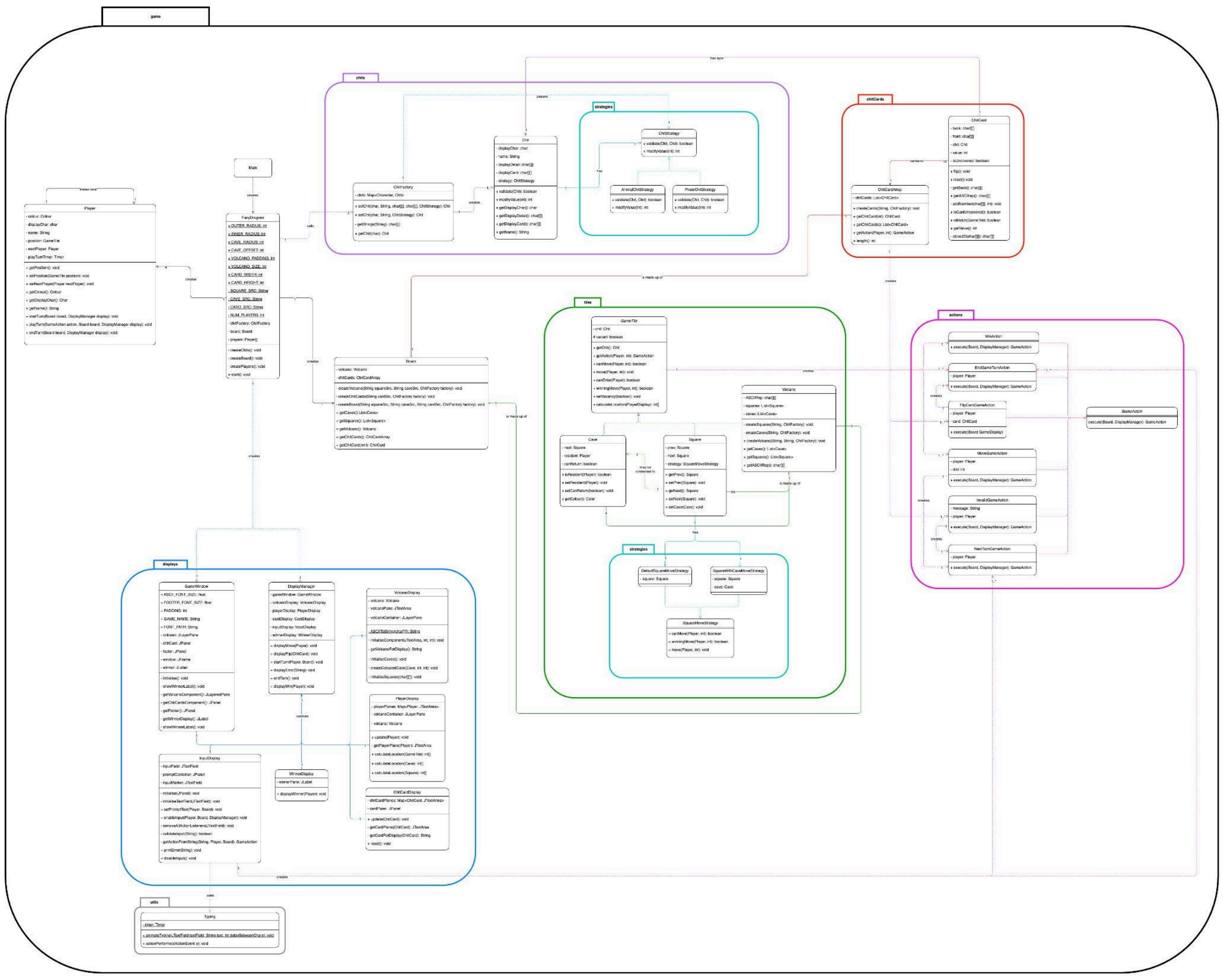
Responsibilities

- createChits(): void
- createBoard(): void
- createPlayers(): void
+ start(): void

Description

FieryDragons is a main class as it runs all of the major initialization of the game, such as the display and the gameboard. Finally it starts the main game launch in a seperate window where the user can interact with it. It also connects to all of the other main classes directly and indirectly through other subclasses.

Class Diagram



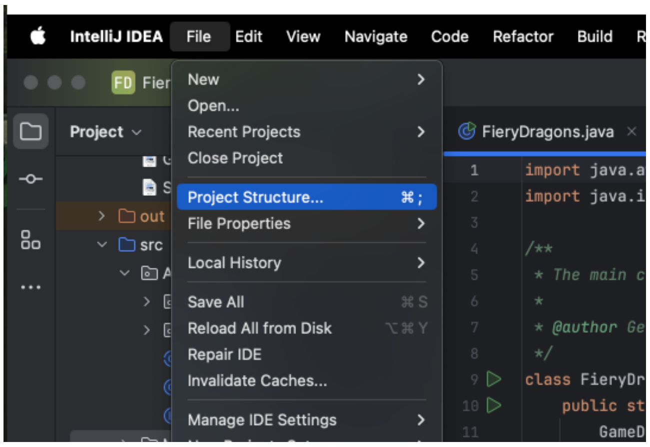
Executable

Executable Description

Opening up the Executable will open a JFrame containing ASCII art displaying the Fiery Dragons volcano on the left, and a spread of Chit Cards on the right. The volcano contains 24 squares, and four caves. 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 from the range of 1 to 16. 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

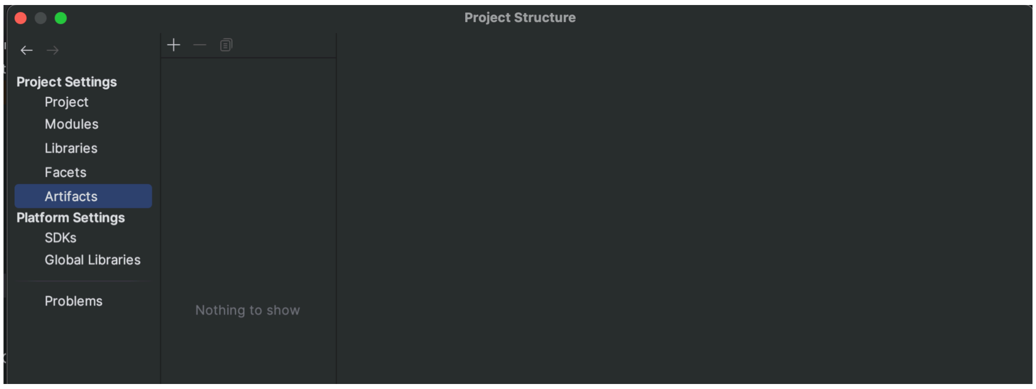
1. After pulling the source code from either Git or through the zip submission, open up the FieryDragons folder in IntelliJ. Make sure you are running Java 22 on a MacOS or Windows system.
2. Mark the “res” folder as the Resources Root.
3. Open the Main file.
4. Go to the menu bar at the top of the screen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



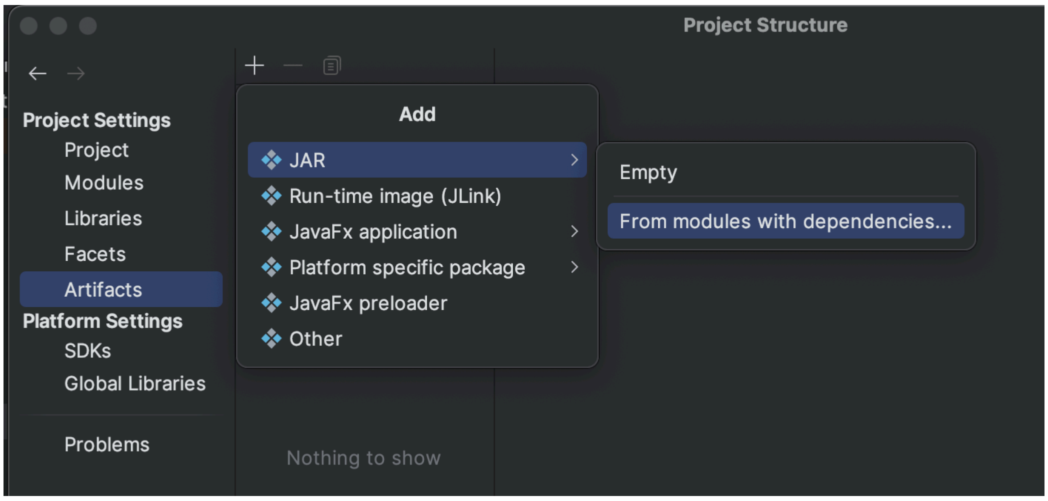
```
import java.a
import java.i

/**
 * The main c
 *
 * @author Ge
 */
class FieryDr
    public st
        GameD
```

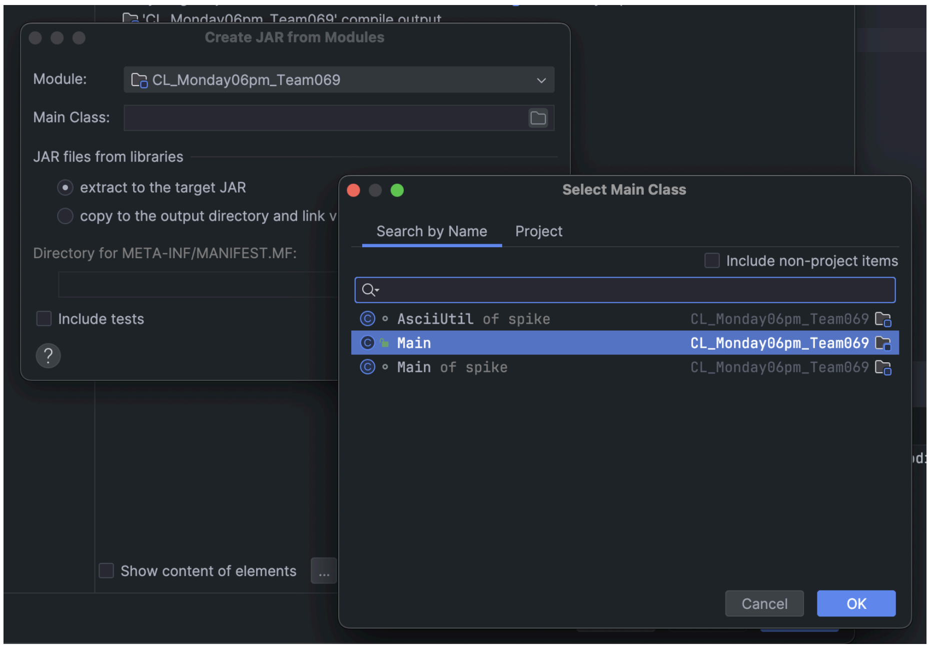
5. A pop-up window should appear. Press Artifacts under Project Settings.



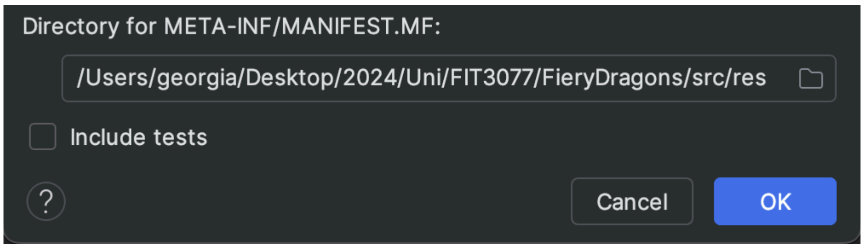
Press the Plus sign, and then add JAR from modules with dependencies.



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

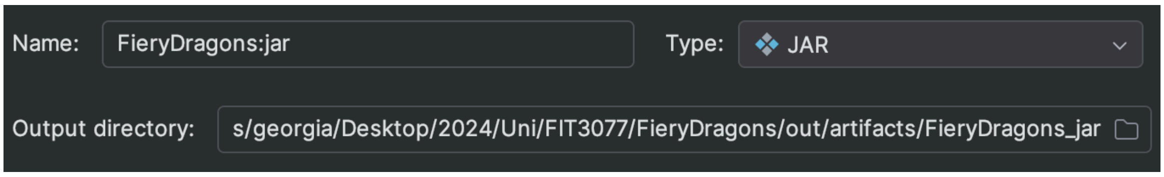

Create JAR from Modules
Module: CL_Monday06pm_Team069
Main Class:
JAR files from libraries
extract to the target JAR
copy to the output directory and link v
Directory for META-INF/MANIFEST.MF:
Include tests
Show content of elements
Select Main Class
Search by Name
Project
Include non-project items
AsciiUtil of spike CL_Monday06pm_Team069
Main CL_Monday06pm_Team069
Main of spike CL_Monday06pm_Team069
Cancel
OK

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



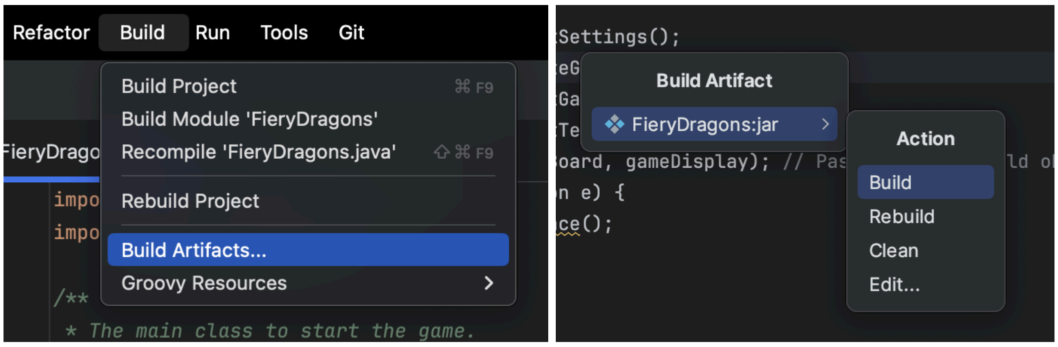
Directory for META-INF/MANIFEST.MF:
/Users/georgia/Desktop/2024/Uni/FIT3077/FieryDragons/src/res

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

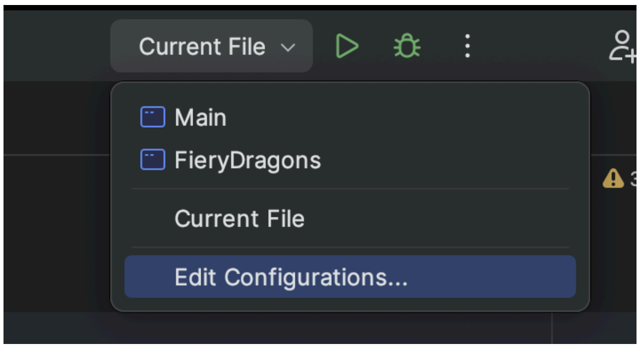


Name: FieryDragons:jar
Type: JAR
Output directory: s/georgia/Desktop/2024/Uni/FIT3077/FieryDragons/out/artifacts/FieryDragons_jar

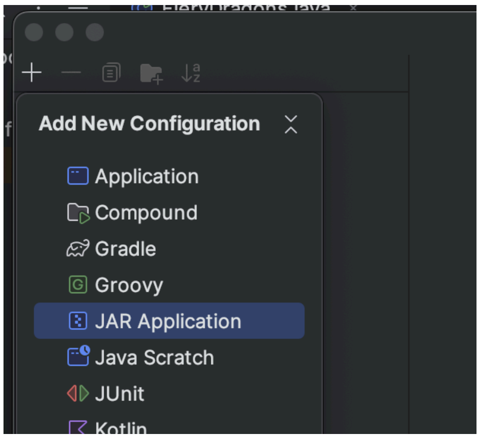
9. Then go to the menu bar at the top of the screen, and select Build to open the drop-down menu. Select Build Artifacts….Then in the new pop-up window, press Build. You should now be able to see the META-INF inside of your res folder.



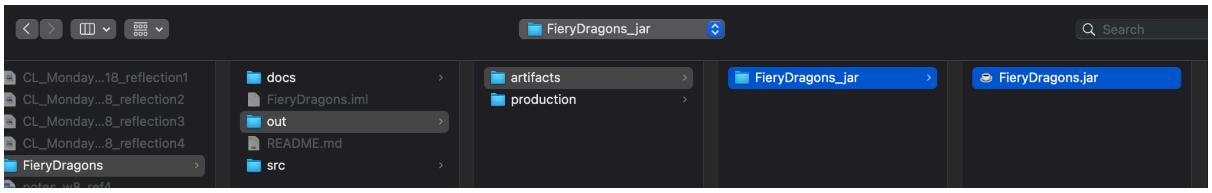
10. Now go to the top right of the IntelliJ window and select Current File, to open the drop-down menu. Then select Edit Configurations…



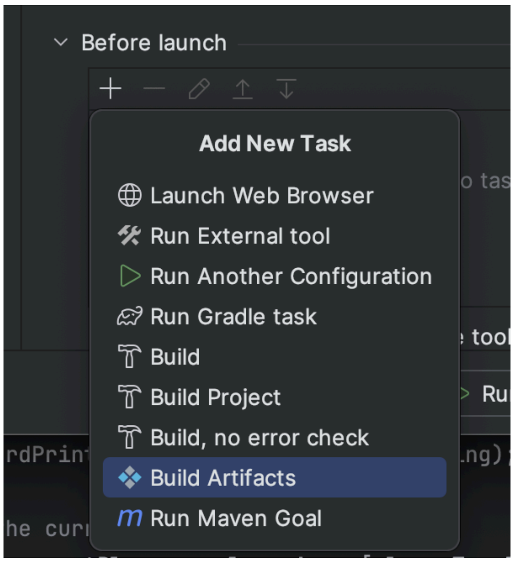
11. A new window should appear named Run/Debug Configurations. Press the plus sign in the top left of the window and select JAR Application.



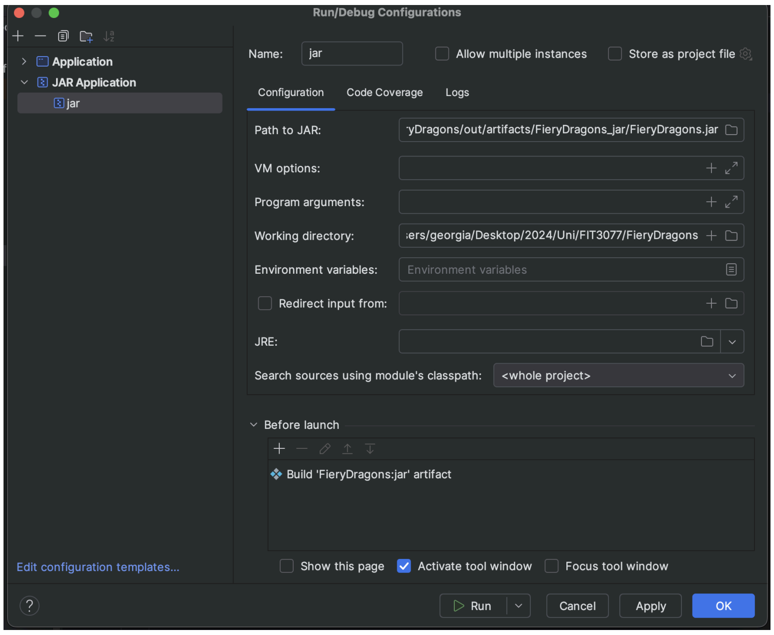
12. Name the JAR Application whatever you like. I have named it “jar” for simplicity. Press the file symbol next to Path to Jar, and select the jar file.



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



14. Your screen should look like this.



Run/Debug Configurations

- Application
- JAR Application
  - jar

Name: jar

☐ Allow multiple instances ☐ Store as project file

Configuration | Code Coverage | Logs

Path to JAR: yDragons/out/artifacts/FieryDragons_jar/FieryDragons.jar

VM options:

Program arguments:

Working directory: ers/georgia/Desktop/2024/Uni/FIT3077/FieryDragons

Environment variables: Environment variables

☐ Redirect input from:

JRE:

Search sources using module's classpath: <whole project>

Before launch

Build 'FieryDragons:jar' artifact

☐ Show this page ☑ Activate tool window ☐ Focus tool window

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.