FIT3077

Software engineering: Architecture and design

Sprint 4 - Documentation

Group name: Master Byter

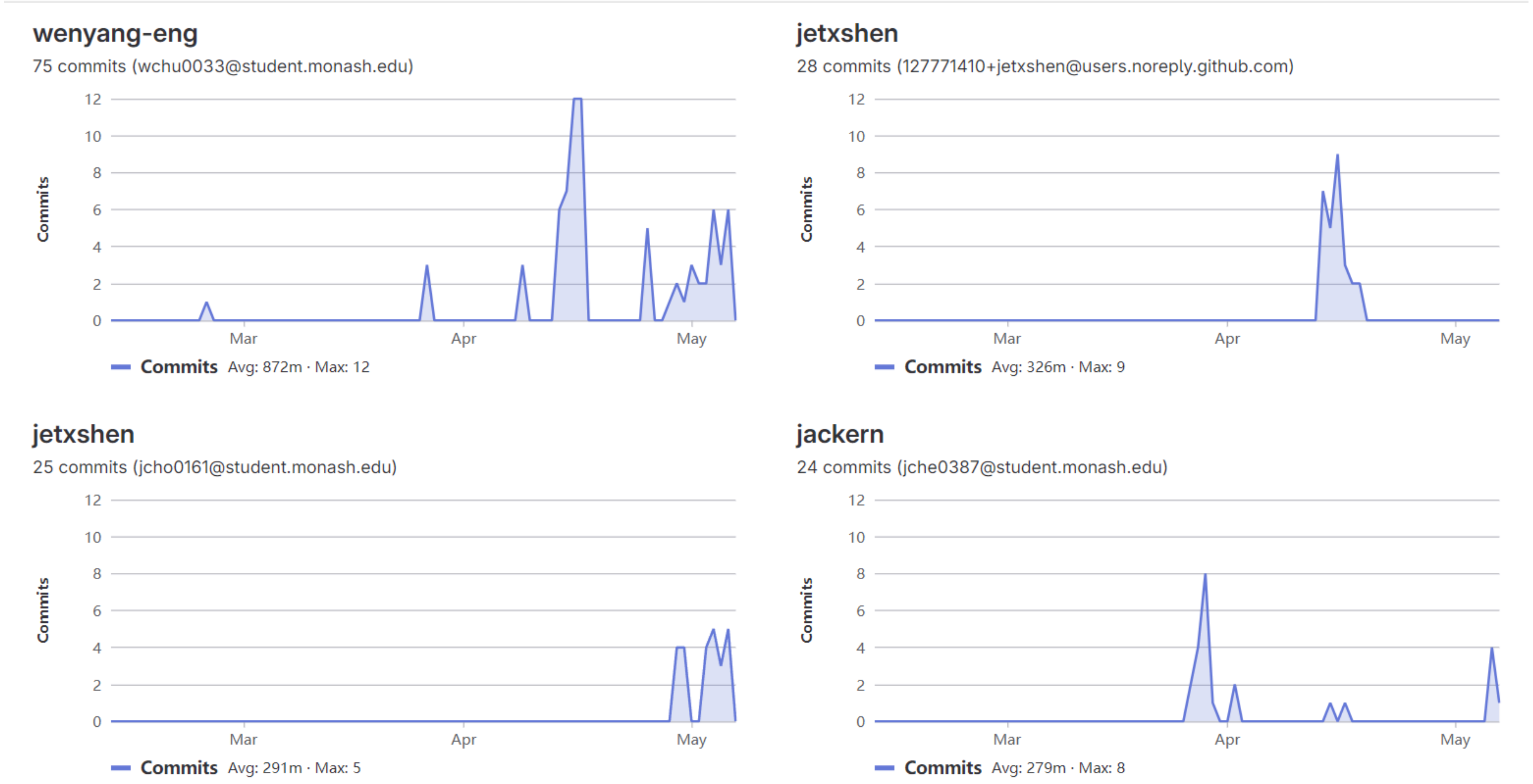
Written by:

Chen Jac Kern

Chong Jet Shen

Chua Wen Yang

1. Screenshot of Contributor Analytics


wenyang-eng
75 commits (wchu0033@student.monash.edu)
Commits Avg: 872m · Max: 12

jetxshen
28 commits (127771410+jetxshen@users.noreply.github.com)
Commits Avg: 326m · Max: 9

jetxshen
25 commits (jcho0161@student.monash.edu)
Commits Avg: 291m · Max: 5

jackern
24 commits (jche0387@student.monash.edu)
Commits Avg: 279m · Max: 8

Figure 1. Screenshot of Contributor Analytics

2. Sprint 4 Extensions

2.1 Required Extensions

2.1.1 New Dragon Card (or “chit” cards)

Chosen New Dragon Card:
New Dragon Card 1

Description/Justification:
The New Dragon Card 1 has the functionality to send the current player to the closest and unoccupied cave which is behind that player. After sending the player to the nearest cave, the player has another chance to flip another dragon card to move forward. However, if the player who flips this new dragon card is at the cave, the turn will directly pass to the next player. For visualisation purposes, four

For how we implement this feature:

1. If any player flips the NewDragonCard at the first time, a completed path of the game will be created (including cave and volcano card).

2. Each player has a path list to move, we find the location of the player based on their path list.

3. Remove all the paths which have been passed by the player from its path list.

4. Find the nearest cave behind the player’s current location with the completed paths created.

5. After the nearest cave has been found, add the paths from the cave to the player’s current location in that player’s path list. After that, send that player to the cave (the first element in the path list).

2.1.2 Loading and saving the game from/to an external (configuration) file in a suitable text (not binary) format.

Description/Justification:

We have implemented a saving and loading function by using binary serialisation using “ObjectOutputStream” and “ObjectInputStream” to save and load the game in a binary format. The save method creates a DataStorage object that holds the game and then serialises it into a binary file called “game_save.dat”. Then, when we load the game, the load method reads the DataStorage object from the binary file and the game components are refreshed to reflect the loaded state. Objects like ChitCardController and VolcanoCardController were not implemented as they are not serialised.

2.2 Self-Defined Extension (1 extension for Team of 3)

Chosen Self-Defined Extension that explicitly addresses a Human Value:
Fiery Dragon's Time Limit Feature

Description/Justification:
Our team realised that as Fiery Dragons is a children's game, these children could potentially get addicted to this game. While it is good for children to have fun, it can be very unhealthy if they were to play the game for extremely long hours.

By looking at the screen for too long, these children will develop many kinds of eye problems like eye fatigue, and potentially blurred vision which could be permanent.

To mitigate such an issue, our team's self-defined extension is to incorporate a Timer in our Fiery Dragon's game. This is by prompting players for their preferred time limit set for each round. We don't expect the game to last too long, so we've limited the time limit to 1-60 minutes.

This will not only reduce chances of these children from developing such eye problems, but also reduce their unhealthy addiction to the game as the screen time allowed on each Fiery Dragon's game round will be set by them.

For how we would handle this feature:

1. Before starting each round, players will be prompted to set a time limit for the current round.

2. A timer will be displayed during the game, indicating how much time left the players have before the game is forced to end.

3. If a player wins the round before the timer runs out, the timer will stop. This is followed by a winning pop up message, giving a choice to start a new round with the same time limit set. Not much changes from Sprint 3 implementation.

4. However, in the event the timer runs out before a player wins the round. The player closest to the end (back to their cave) will be the winner. The same winning pop up message will appear, and the player has a choice to start a new game with the same time limit set.

3. Object-Oriented Design

3.1 Revised class diagram

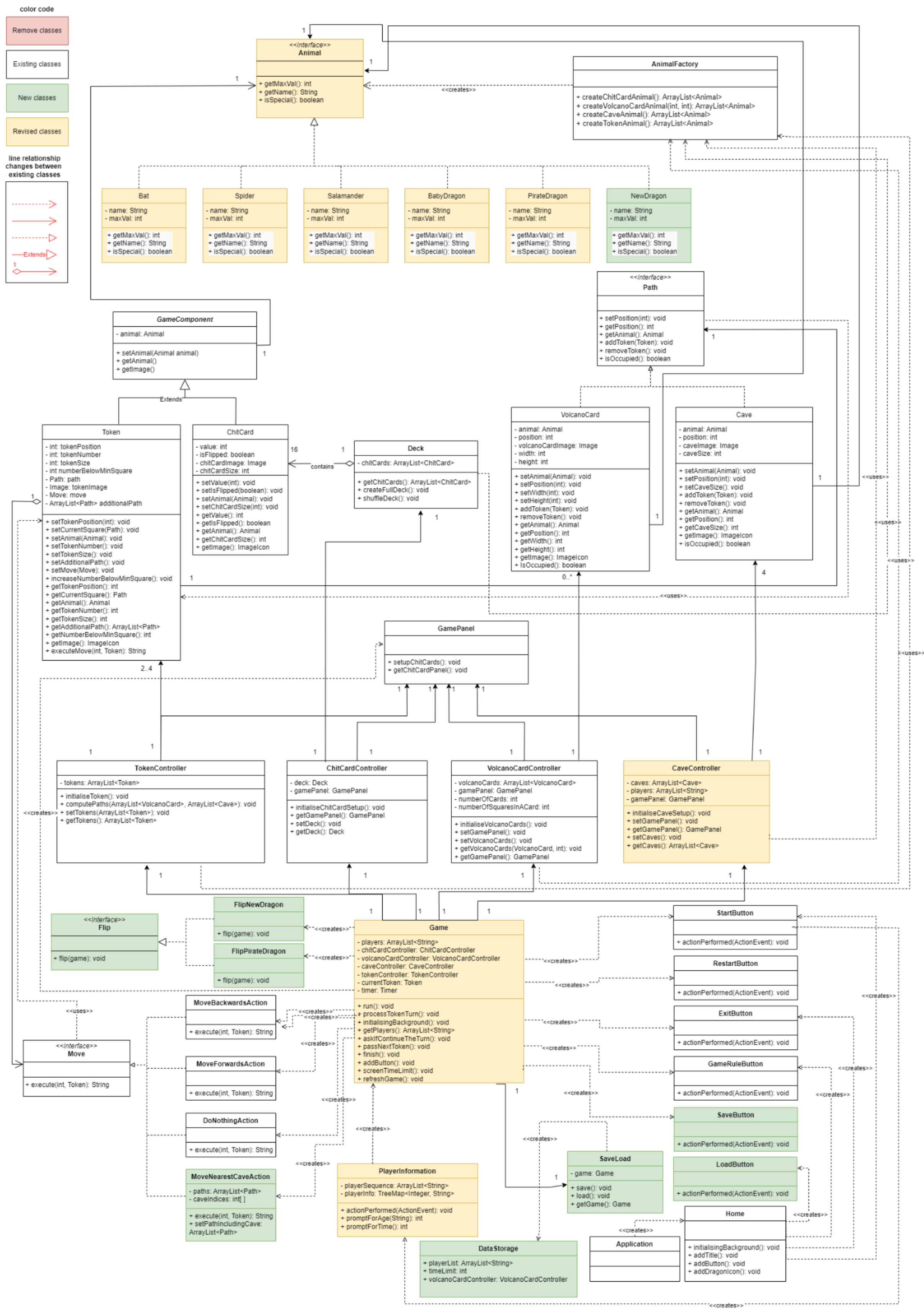


Figure 2 Revised UML class diagram

Based on the feedback given in Sprint 3, we have revised the CaveController class to allow randomised positions of the caves in the game. We've also considered where only one cave can be within each volcano card which has 3 tiles each.

To implement the new dragon card, where we have chosen New Dragon Card 1. We've decided to use abstraction on the flipping of dragon cards with a Flip interface. Where we've also added FlipPirateDragon and FlipNewDragon classes to separate the responsibilities of each type of dragon card functionality into different classes. This is to avoid the functionality to be implemented within the Game class (avoiding god class). Fortunately, the addition of these two classes satisfies the Dependency Inversion Principle (DIP) as they implement the Flip interface. This also adheres to the Open Closed Principle (OCP) since we are able to add other new types of dragon ("chit") cards without affecting other existing classes.

Every animal class implemented with the Animal interface has also added a new isSpecial() function. This to help the game determine if the type of dragon card flipped is special or not. Our definition of special would mean movement of player's token

For us to implement the loading and saving game feature, we have created two new buttons (SaveButton and LoadButton) for us to save the current game and load the current game at a later time. No abstract class or interface is needed to be created as this abstraction is already handled by the in-built ActionListener interface. These buttons utilise the built-in ActionListener interface, which handles the button click events, negating the need for additional abstract classes or interfaces. Besides, the SaveLoad and the DataStorage classes are used to save, load, and store data respectively, this ensures that the SRP is fulfilled because game data which needs to be saved is separated from the game itself.

To implement our self-defined extension of the time limit feature per round, a new Timer attribute along with a screenTimeLimit function has been added within the Game class. The reason for not separating these additions into a separate class is because these new additions work very closely within the Game class. The Game class is responsible for the whole game itself, which would include a time limit within each game. PlayerInformation class was also revised to prompt players for their preferred time limit. No new class was needed as the time limit per round is also information needed by players. Hence, Single Responsibility Principle (SRP) is retained.

In the current design, we have retained the use of three design patterns: FactoryMethod (creational), Model, View, and Controller pattern (structural), and Strategy pattern (behavioural). FactoryMethod is used in the AnimalFactory class to create animals, based on the Animal interface, for each game component. Within this sprint, an additional NewDragon animal is created for the New Dragon Card 1 extension.

For the MVC pattern, no new game components have been added to our Model. GamePanel remains as the View to have all game components displayed. And all controllers for each game component handles the new dragon card extension without modifications needed.

Finally, for the Strategy pattern, we have added a new action class (MoveNearestCaveAction) to be the new distinct strategy, under the Move interface, to facilitate the movement of the token when New Dragon card 1 is flipped to move backwards to the nearest empty cave. This pattern has also been used for the Flip interface, where there are two concrete strategies (FlipPirateDragon, FlipNewDragon) to facilitate the flipping of these unique dragon cards.

4. Reflection on Sprint 3 Design

4.1. Extensibility of our design/implementation

In the Sprint 3 design, we followed the SOLID principles to ensure the extensibility and reusability. For Single Responsibility Principle (SRP), each game component is created as a different class to perform their own responsibilities. This ensures the modularity of the software, and hence improves the extensibility of the design because modifications on a component will not affect the other. For instance, the movement action of the token is separated to MoveForwardsAction and MoveBackwardsAction. Besides, ChitCard, VolcanoCard, Cave, and Token are also implemented as different classes, therefore extension on one of them will not affect the other.

Besides, we utilised the abstraction layer (interface) to ensure the Open Closed Principle (OCP) and the Liskov Substitution Principle (LSP) are adhered to in the design. With these principles, adding new features without modification on the existing class became easier. For example, in the current design, VolcanoCard and Cave implement the Path interface. When the ArrayList of Path interface is used to compute the paths of each token, the paths include VolcanoCard and Cave because of polymorphism, which follows LCP. Additionally, the VolcanoCard and Cave have different implementations of getImage(), this ensures that the OCP is followed.

The Move interface in the design also adheres to the Dependency Inversion Principle (DIP) and Interface Segregation Principle (ISP) because it is the attribute of the Token class, where the token class can call the execute() method in the MoveForwardsAction and MoveBackwardsAction. The insulated changes between the high-level and low-level code increases the extensibility of the system design.

The use of the design pattern also facilitates the extensibility and reusability in the design. One of the examples is that the Model, View, and Controller (MVC) pattern, the ChitCard, VolcanoCard, Cave, and Token becomes the Model class which stores the data for the game component. The Controller classes corresponding to them facilitate the logic of setting up their positions and information required (e.g. computePaths of the Token). While the View depends on the Game class to display them out. This separation of concern also helps in extensibility of the system because we can modify the Model, View, or Controller without affecting the other parts of the MVC.

However, there might be some improvements that can be made in the current design. One of them is that we could apply the software architecture to the design in a better way. For example, we can use the layered architecture to differentiate the presentation layer and the logic layer in the Game class. This is because the GUI and the game logic are included in the current Game class. Creating the GameGUI and GameLogic classes, then calling them in the Game class would be a better solution to separate the responsibilities and concerns.

4.2. Level of difficulty for each of the incorporated extensions

When it came to creating the new Dragon ("chit") card, where we have chosen New Dragon Card 1. It was relatively easy to extend this new dragon card because the design pattern implemented in the Sprint 3 facilitates the extensibility in the design. For instance, the Factory Method used (AnimalFactory and Animal interface) allows the NewDragonCard to implement the same getName() and getImage() method with different implementations. Apart from that, the Strategy pattern applied to the movement-related classes with the Move interface. This implementation makes the extensions of moving the token to the nearest behind the cave more easier, since different movements would be made based on the flipped card in the Game class.

However, in Sprint 3 design, we only consist of the card which is different from others - PirateDragon. This leads to the code smell (if-else statement) in our design, for checking if the flipped card is the pirate dragon with the animal name. Therefore, in the future, we would avoid this problem by applying suitable design patterns or adding abstraction layers. This is because these approaches will promote encapsulation which allows reuse of code and hence reduces redundant code. For instance, to avoid further code repetition in Sprint 4, we will apply a Strategy pattern with the hashMap<animal name, Flip object> flipMap to differentiate pirate dragon and new dragon cards.

For the save load feature, it is relatively difficult to implement because there are various attributes to save and load within the game object. For instance, the Controller objects, players list, GameComponent objects, the attribute of the GameComponent objects within the Game class should be stored because they are the attributes of the Game object in Sprint 3. Therefore, saving a game object would affect the implementation of classes which are attributes of it, since most of the attributes are non-serializable data.

To avoid this issue in the future, we should consider reducing coupling and improve cohesion during the design phase. One of the approaches we thought of include creating more abstraction layers or inheritance in the class. This is because we can utilise the interface or superclass as our attributes, and hence reduce the complexity of the class. If we require a specific method in the children's class, we could use polymorphism to implement it. Another strategy to improve it is to avoid multiple chain calling. If the method is called a lot of times from another class, it will be better to consider putting this method to the class which always calls it. For example, if game.getVolcanoController.getDeck() is called many times from the Game class, we can change the Deck class as the attribute in the Game class. This will reduce the number of non-serializable attributes in a class, and only essential attributes will be included in the class, so saving the attribute will be easier.

When it came to our self-defined extension, where players set their preferred time limit for each round. This was relatively simple for the PlayerInformation class from Sprint 3 to implement this extension, since a new method of promptForTime, which is similar to the promptForAge can be used to implement this feature.

However, the extensions can be made more convenient if the logic and GUI can be differentiated from the Game class in Sprint 3. Hence with the time limit label can be added in GUI class, while the logic of finding the player who moves more steps can be done in the logic class. With this practice, extensions made on GUI or game logic will become easier since both do not affect the others. Therefore, this alerts us that in future practice, the frontend (GUI) and the backend (logic) should be separated clearly to enhance the extensibility of the system.

5. Executable

5.1 Description of executable

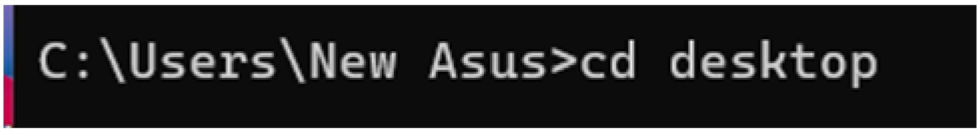
Windows is the main platform for the executable file. Before opening and running the file, you would like to have Java SE Development Kit 22.0.1 and OpenJDK-22 (Oracle OpenJDK version 22.0.1) installed on your machine.

Here is the website link for installation:

https://www.oracle.com/my/java/technologies/downloads/#jdk22-windows

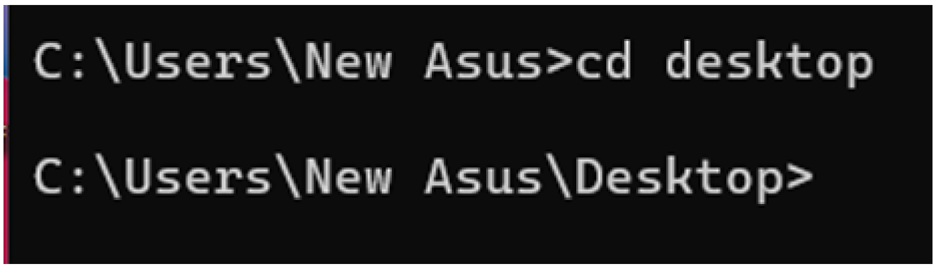
5.2 Instructions of running executable

1. Download the sprint4.jar zip folder.
2. Unzip the folder then copy the jar file to the desktop.
3. Open the command prompt and type: cd desktop
   For example:



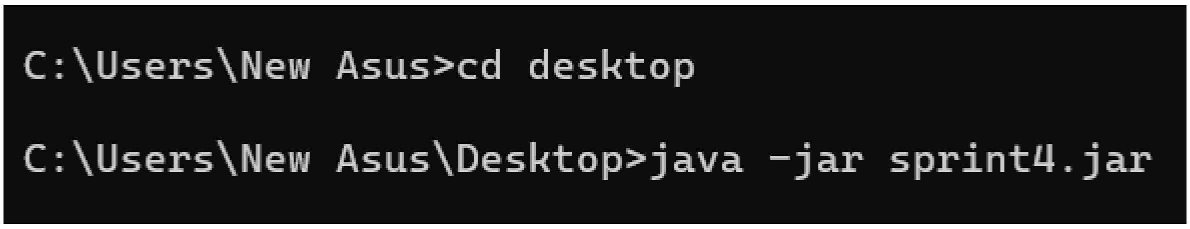
```
C:\Users\New Asus>cd desktop
```

4. Press Enter, you should find that on the next line, there is "\Desktop"



```
C:\Users\New Asus>cd desktop

C:\Users\New Asus\Desktop>
```

5. Then continue to type: java -jar sprint4.jar



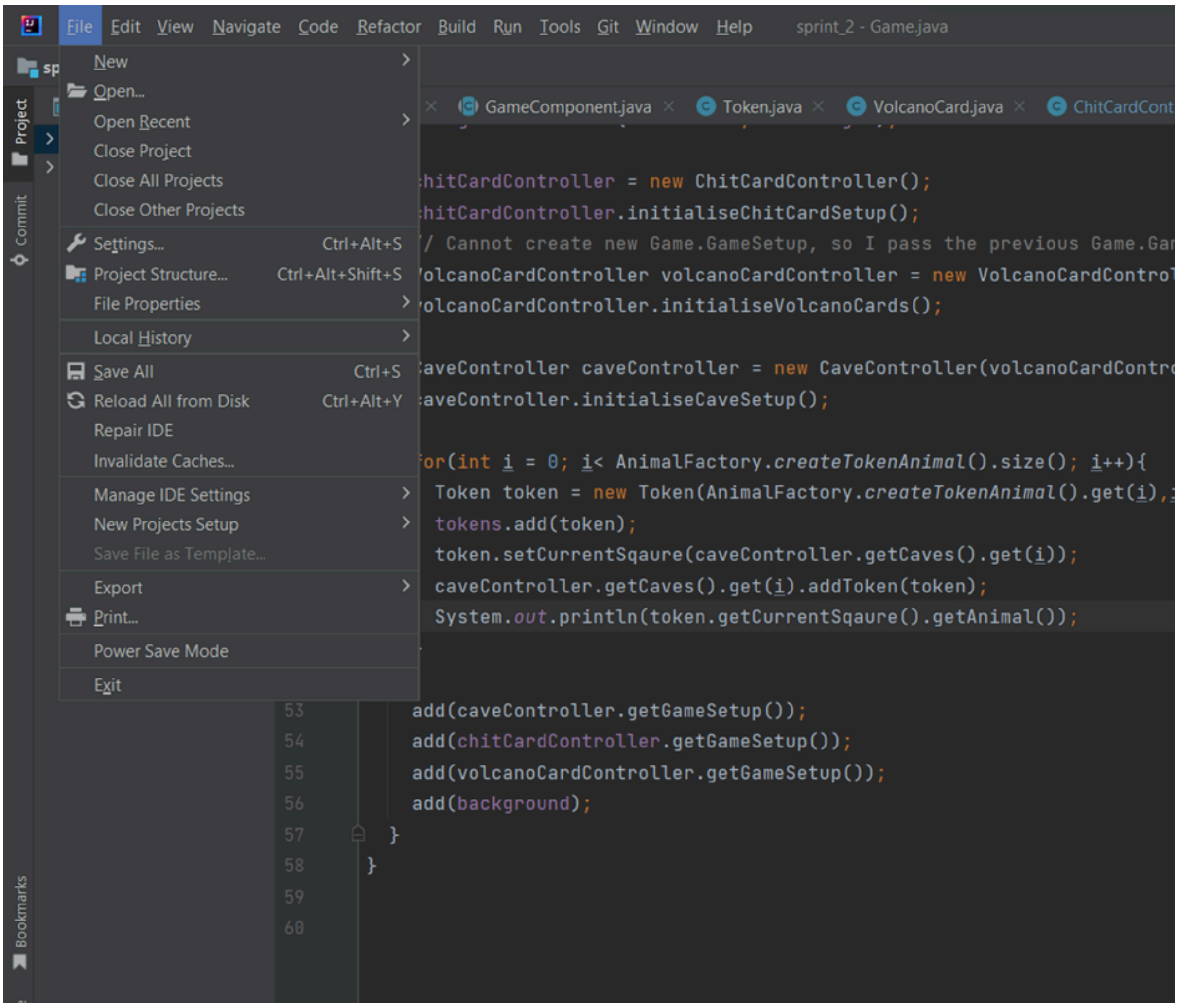
```
C:\Users\New Asus>cd desktop

C:\Users\New Asus\Desktop>java -jar sprint4.jar
```

6. Press Enter again, the game should be open.

5.3 Instructions of building executable

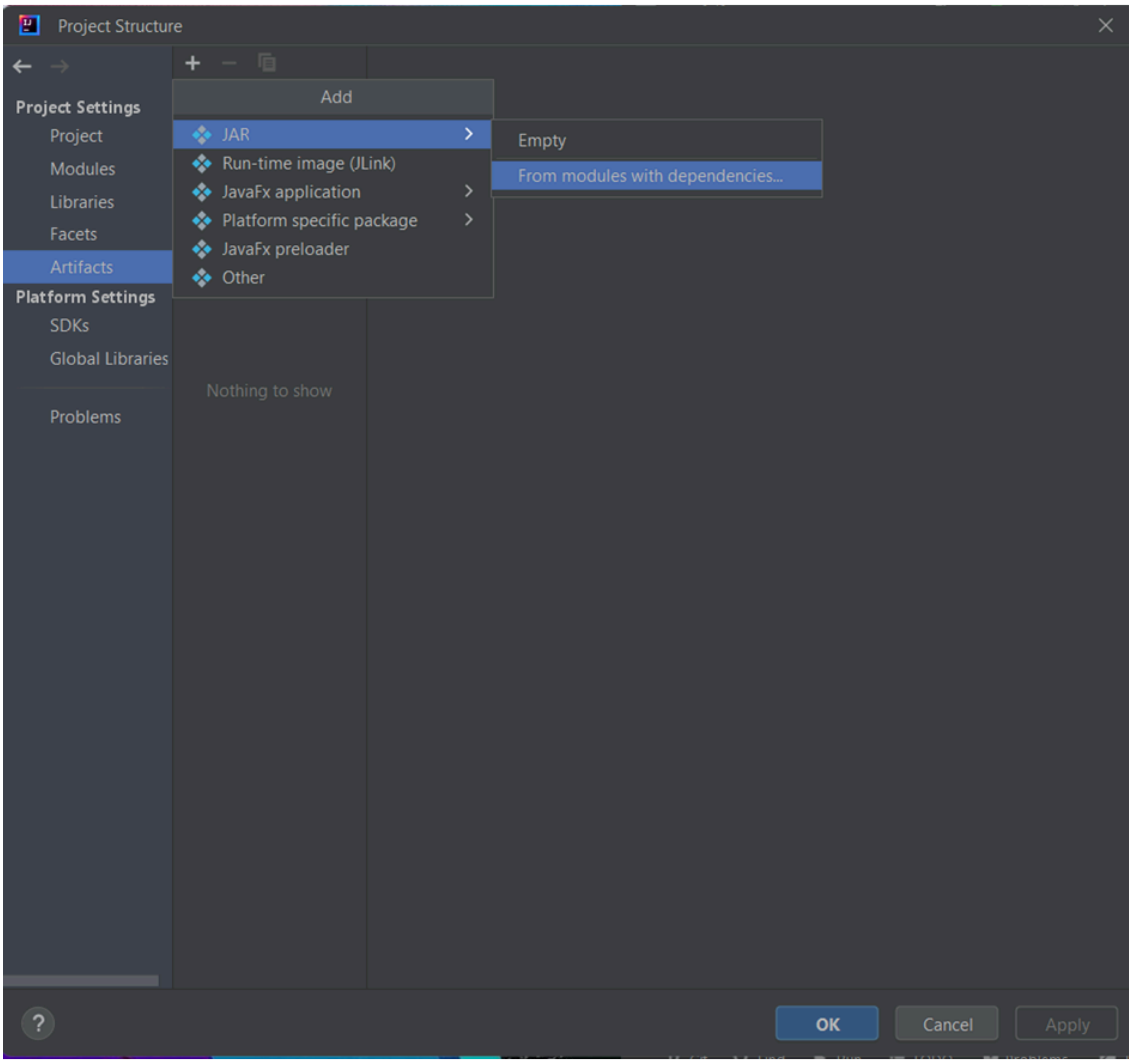
Our team applied IntellijIDEA as the IDE to create the project, here are the steps to build the executable .jar file:

1. First, click on the File -> Project Structure…

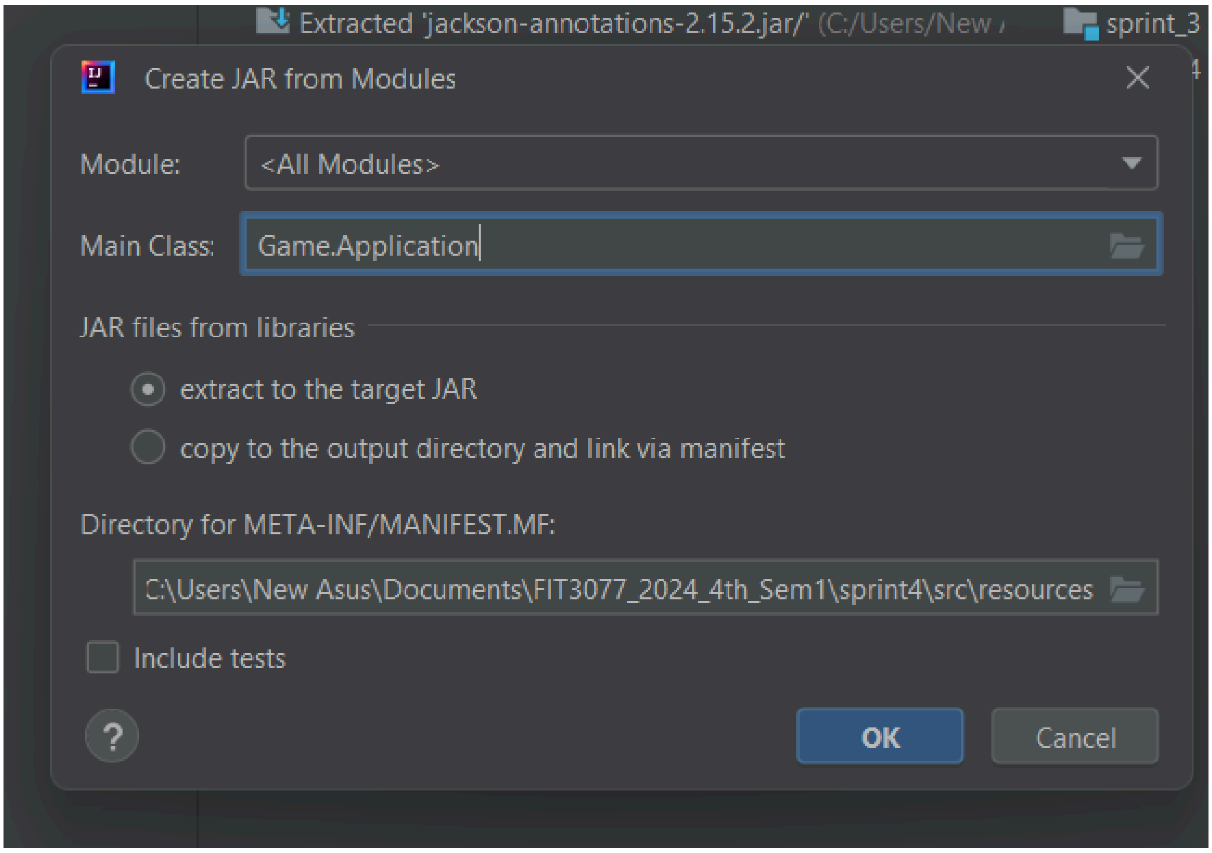


```
hitCardController = new ChitCardController();
hitCardController.initialiseChitCardSetup();
/ Cannot create new Game.GameSetup, so I pass the previous Game.Ga
olcanoCardController volcanoCardController = new VolcanoCardContro
olcanoCardController.initialiseVolcanoCards();

aveController caveController = new CaveController(volcanoCardContr
aveController.initialiseCaveSetup();

or(int i = 0; i< AnimalFactory.createTokenAnimal().size(); i++){
 Token token = new Token(AnimalFactory.createTokenAnimal().get(i),
 tokens.add(token);
 token.setCurrentSqaure(caveController.getCaves().get(i));
 caveController.getCaves().get(i).addToken(token);
 System.out.println(token.getCurrentSqaure().getAnimal());


    add(caveController.getGameSetup());
    add(chitCardController.getGameSetup());
    add(volcanoCardController.getGameSetup());
    add(background);
  }
}
```

2. Next, navigate from “Artifacts” -> click on “+” signal -> select “JAR” -> “From modules with dependencies”, then click Apply -> OK.



3. Choose the main class, and click OK.



Extracted 'jackson-annotations-2.15.2.jar/' (C:/Users/New

sprint_3

Create JAR from Modules

Module: <All Modules>

Main Class: Game.Application

JAR files from libraries

[✓ extract to the target JAR]

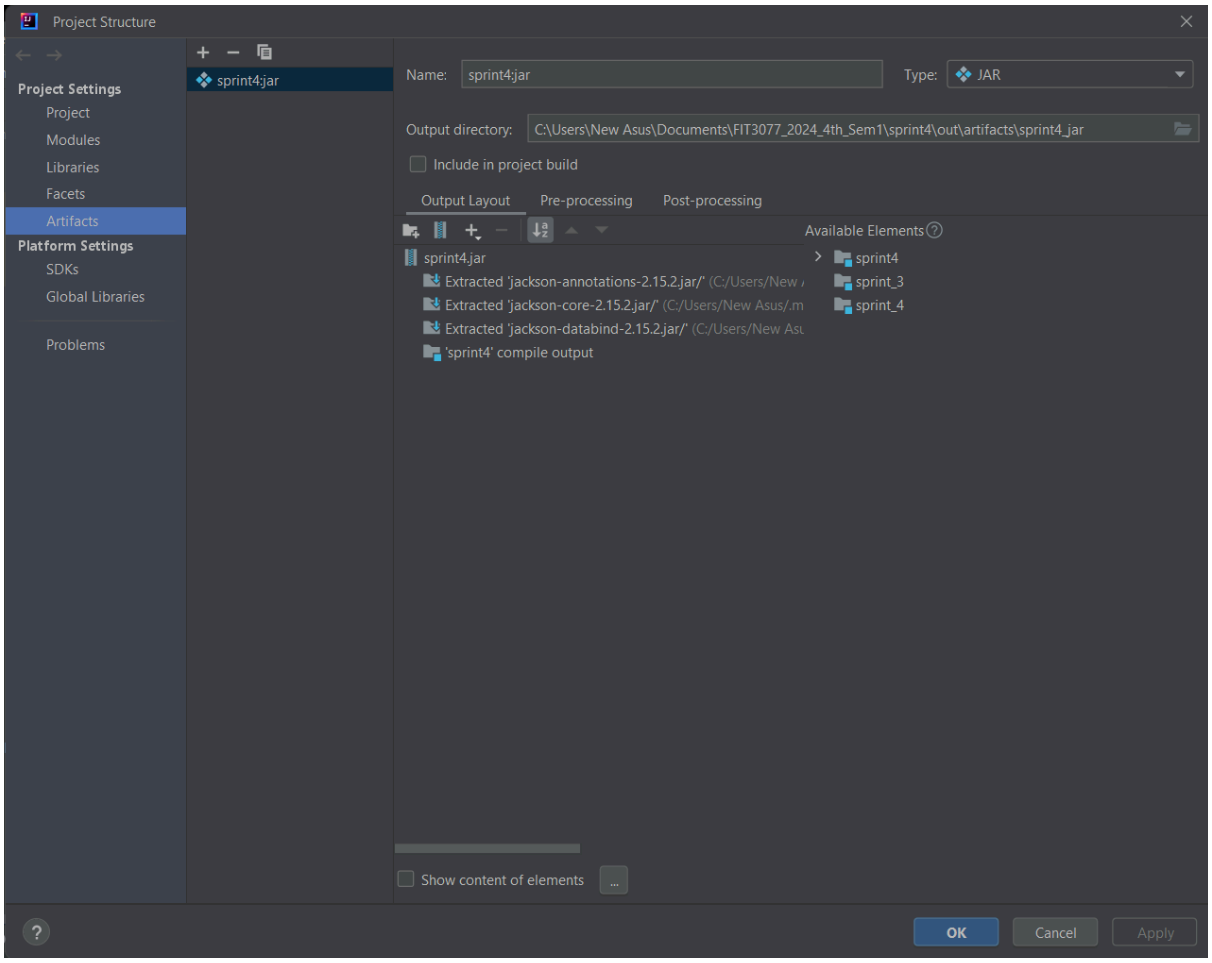
copy to the output directory and link via manifest

Directory for META-INF/MANIFEST.MF:

C:\Users\New Asus\Documents\FIT3077_2024_4th_Sem1\sprint4\src\resources

Include tests

4. Navigate to Apply -> OK.


Name: sprint4:jar

Type: JAR

Output directory: C:\Users\New Asus\Documents\FIT3077_2024_4th_Sem1\sprint4\out\artifacts\sprint4_jar

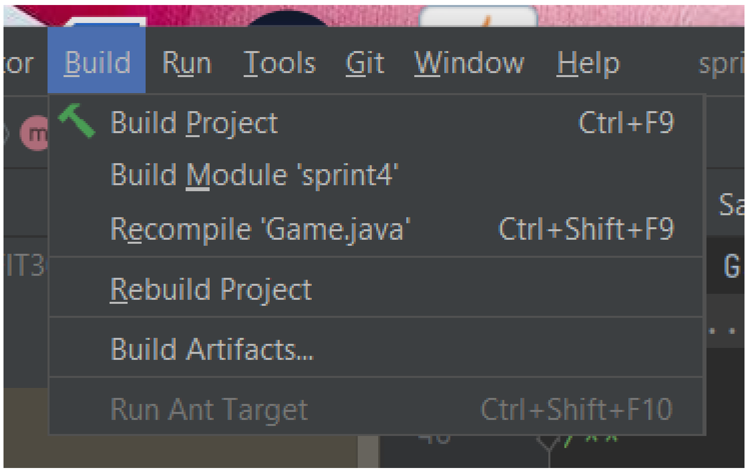
Include in project build

sprint4.jar
- Extracted 'jackson-annotations-2.15.2.jar/' (C:/Users/New
- Extracted 'jackson-core-2.15.2.jar/' (C:/Users/New Asus/.m
- Extracted 'jackson-databind-2.15.2.jar/' (C:/Users/New Asu
- 'sprint4' compile output

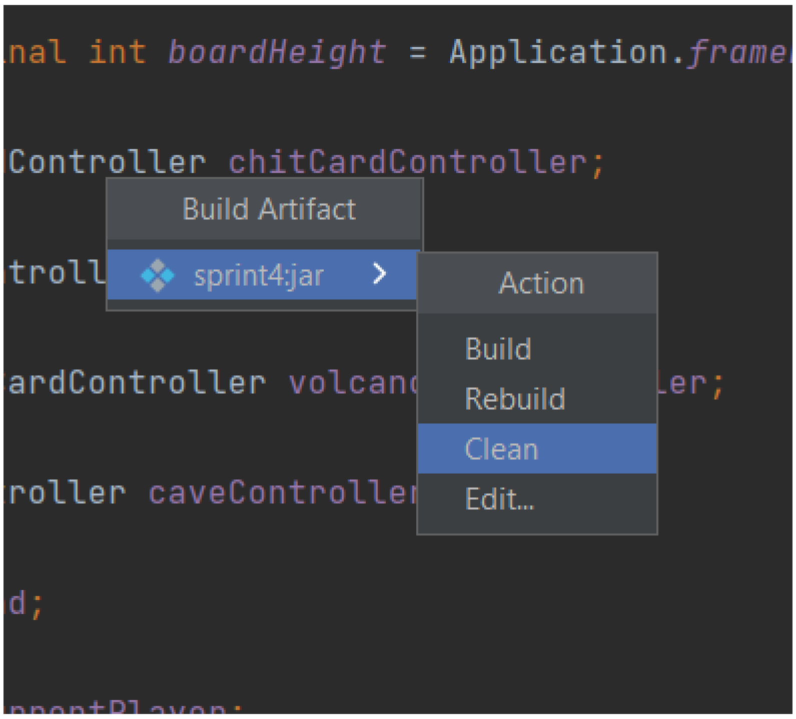
Available Elements
- sprint4
- sprint_3
- sprint_4

Show content of elements

5. Next, click on “Build” -> “Build Artifacts”.

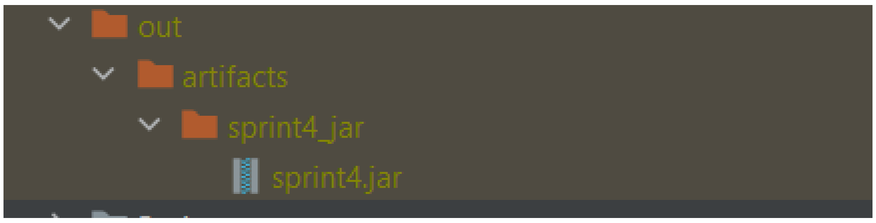


6. Click on “Build”.



```
nal int boardHeight = Application.frame
Controller chitCardController;
troll
ardController volcan         ler;
roller caveControlle
d;
```

7. Then you can see the -jar file (executable) in the directory.



6. Additional Notes

Here are some links for Sprint 4 for reference:

1. Sprint Contribution Log:
   https://docs.google.com/spreadsheets/d/10OCArWbF7X28P6jE5t_A_OGyh01ALSsA
   VlzDX1IH_jQ/edit#gid=0

2. Draw.io for UML diagram:
   https://app.diagrams.net/#G1Caga46tX6Dy95ySHGej2UAnR6ovqRSjE#%7B%22pa
   geId%22%3A%22Tn4xBR2LgAawVbLnZKPs%22%7D