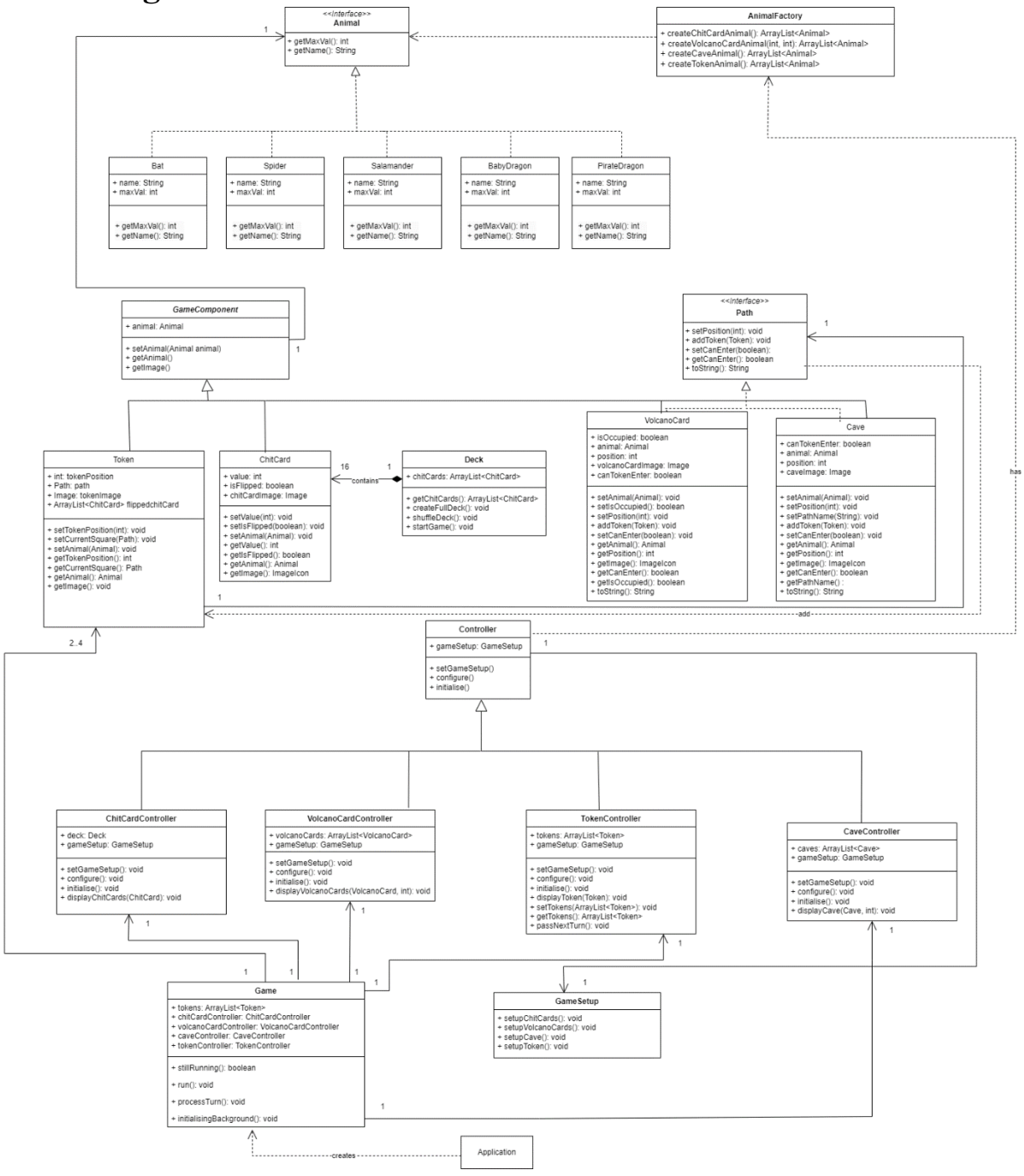
FIT3077 Sprint 2

Name: Chua Wen Yang

Student ID: 33274320

1.0 Object-Oriented Design and Design Rationale

1.1 UML Diagram



1.2 Design Rationale

1.2.1 Key Classes:

- Game:
  The Game class has the responsibility of initialising the game, running the game, and changing the turn of the player while running the game. It is more appropriate to make it as the class because it acts as the main handler of the Fiery Dragon game, using a method for it will make that method overcomplicated.

- GameSetup:
  The GameSetup class has the responsibility of the game board setup (e.g. chit cards, volcano cards, tokens, and caves). It is more appropriate to make it as a class because of the adherence to the Single Responsibility Principle, meaning that GameSetup has only the responsibility of making the setup.

- AnimalFactory:
  The AnimalFactory class has the responsibility of creating the animals of chit cards, volcano cards, tokens, and caves with static methods. It is more appropriate to make it as a class because it can allow the reusability of the code, for example, if different components require animal creation, the AnimalFactory class can be reused in each of them.

- Controller, ChitCardController, VolcanoCardController, CaveController, TokenController:
  These controller classes are created for managing the configuration and initialization of volcano cards on the game board. It is more appropriate to keep them as classes due to the benefits of maintainability and readability that come from consolidating related functionalities within a single unit.

1.2.2 Key Relationships:

- Cave and VolcanoCard implements Path:
  Considering the tokens can stand on the volcano card and the cave components, these two classes will implement the Path interface because this will define the contracts with the method implementation between them. However, it provides sufficient flexibility on how the Cave and VolcanoCard will implement these methods. Another alternative would be these two classes inherit the Path abstract class. Unlike interface, some concrete methods in the abstract class do not require subclasses to implement them. After comparison, I decided to choose the interface because the Path should not have additional methods for itself, while it is unsure whether the implementations of the two classes are different. Therefore, the relationship of the Cave and VolcanoCard implements Path interface is chosen.

1.2.3 Decisions around Inheritance:

Alternative A: Make GameComponent become an abstract class, the VolcanoCard, ChitCard, Token, and Cave inherit it.	Alternative B: Not creating the GameComponent class, VolcanoCard, ChitCard, Token, and Cave do not inherit from it.
Advantages: o Allow polymorphism between the GameComponent and its subclasses. o Promote reusability of the design. o Align with the Liskov Substitution Principle (LSP). o The hierarchy between the GameComponent and its subclasses makes the design more organised.	Advantages: o The complexity of the design will be reduced.
Disadvantages: o Increase the complexity of design.	Disadvantages: o Might have two same implementations in two classes. o The design might be unorganised because VolcanoCard, Token, Cave, and ChitCard classes might scatter across the codebase.
Decision: Alternative A. The advantages of A outweigh the disadvantages, especially when it comes to polymorphism which the VolcanoCard, ChitCard, Token, and Cave might have different implementations of initialise and configure methods, while Alternative B has fewer advantages.	

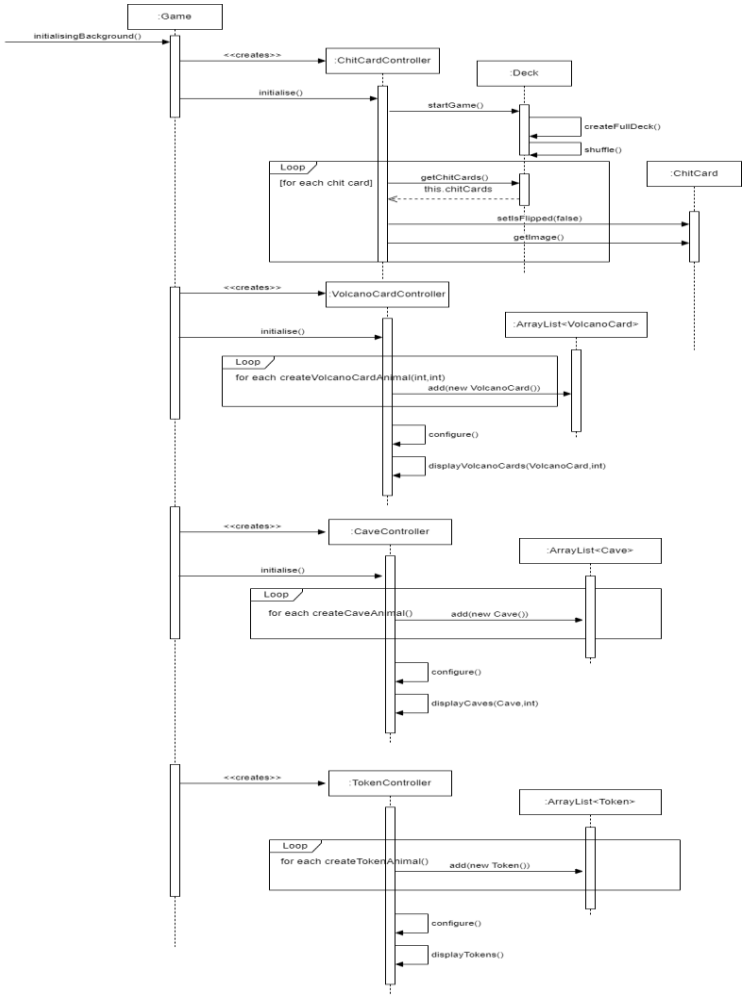
1.2.4 Cardinalities:

- **1 Deck contains 16 ChitCard:**
  The Deck class creates all the chit cards that will be used in the Fiery Dragon game. There are a total of 16 chit cards, therefore the cardinalities are 1 deck containing 16 chit cards.

- **1 Game has 2...4 Token:**
  The Fiery Dragon game allows 2 to 4 players in the game, so there will be at least 2 tokens and at most 4 tokens in the game, therefore the cardinalities are 1 game has 2 to 4 tokens.

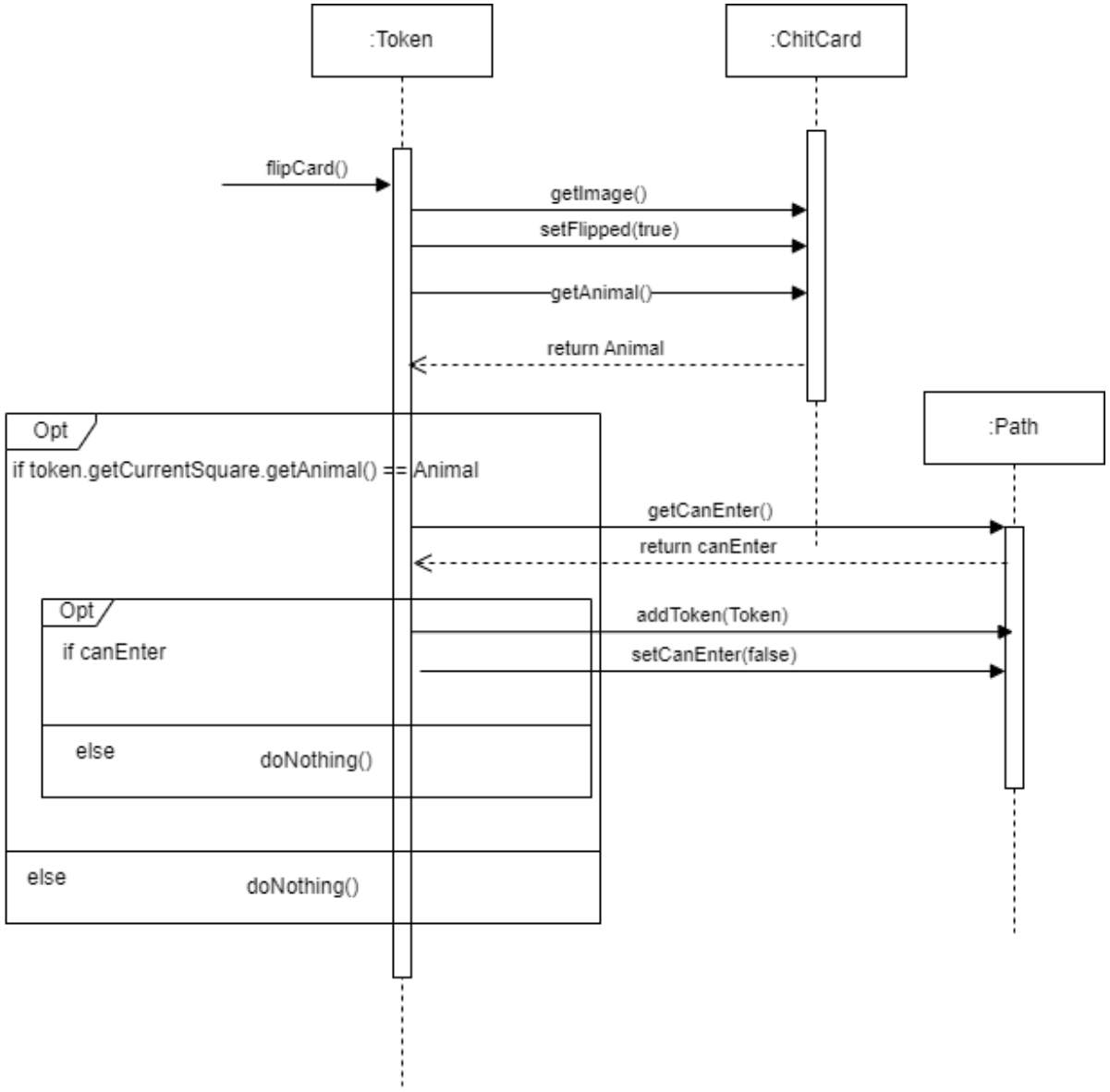
1.3 UML Sequence Diagram with 5 key functionalities

1.3.1 Key Functionality 1: Initial Game Board Setup



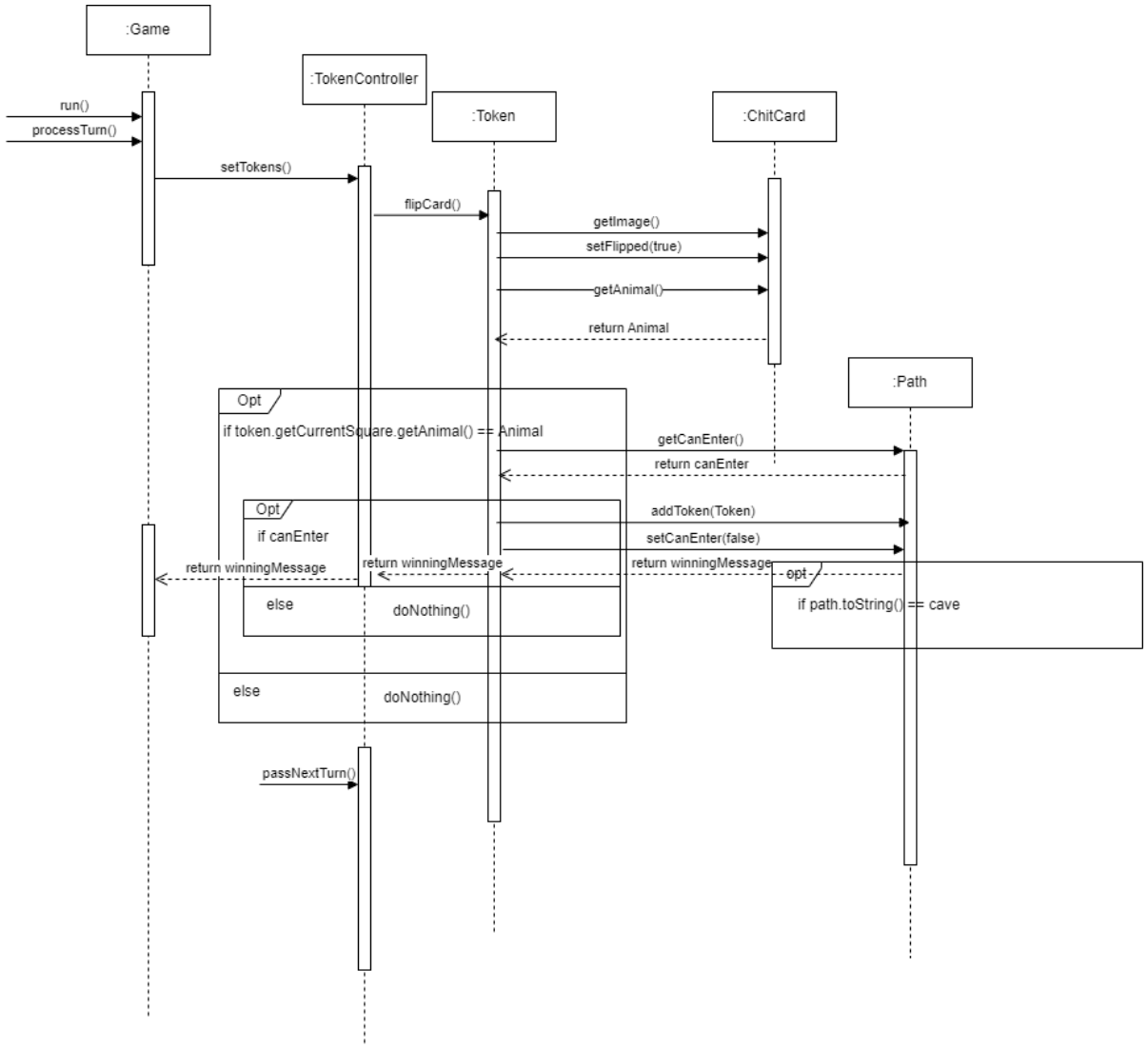
The sequence diagram above is for key functionality 1: Set the initial game board. The Game class will create four Controller classes (ChitCardController, VolcanoCardController, CaveController, and TokenController) respectively. ChitController will start the game by creating the chit cards and shuffling the cards, then get the images from each ChitCard object. While VolcanoCardController, CaveController, and TokenController will create a list of volcano cards, tokens, and caves facilitated with the AnimalFactory.createVolcanoCardAnimal(), .createTokenAnimal(), and .createCaveAnimal() method respectively. Then those cards and tokens will process configure and display their images with configure() and display().

1.3.2 Key Functionality 2&3: Flipping chit card & Move Token



When the token flips the chit card, it will show the image of the chit card with the getImage(), then set the flipped status of that chit card to true. Then get the animal on the chit card, and check if the animal on the chit card is the same as the animal on the square where the current token is located. If this is true, continue to check if the square that the token will move to can enter or not. If not, the token will do nothing and stay the same square.

1.3.3 Key Functionality 4&5: Change of the Player's turn and Winning Situations



When the Game class implements run() and processTurn(), the TokenController sets each token (player), then the first token flip the chit card with flipCard() method. Then the rest of the method is the same as the key functionality 2 and 3. If the returned string from the toString method of the Path is "cave", the winning message is returned to inform that there is a winner. Otherwise, it will pass to the next token (player) to continue to flip the card and do the rest as the same as the first token (player).

1.4 Design Patterns

1.4.1 Factory Method

I applied the Factory Method to create the list of the Animals on the ChitCard, VolcanoCard, Cave, and Token. This is because it saves system resources by just reusing the existing objects. For example, from my AnimalFactory, the method of creating the animal list is reusable since it is static and consistent. Besides, using the AnimalFactory as the creator class also ensures the design follows the Single Responsibility Principle and Open/Closed Principle as it is only responsible for creating the animals while also allowing the introduction of new types of products into the programs without affecting the code. I have also considered Abstract Factory, however, it turned out that numerous classes were created, so I finally decided to use the Factory Method.

1.4.2 Model View Controller (MVC)

Besides, I also used the MVC design pattern in my design solution to operate the game board components. In my design, the models are Token, ChitCard, VolcanoCard, and Cave. The view is the GameSetup. The controllers are TokenController, ChitCardController, VolcanoCardController, and CaveController. The client class is the Game class.

The GameSetup object is first passed into the Controller class to process the logic of showing the image of the models. The Controller class also get the attribute (data) from the models such as Image to do further processing. Then the Game class can just call the controllers to initialise the background.

The reason I used MVC is that it will help the design to achieve the separation of concerns, which will make the whole structure organized and also enhance maintainability. Besides, the clear separation also improves the extensibility of the design, for example, adding new game components. The possible drawback of it is that it might increase the complexity of the design. However, extensibility and maintainability are more important aspects when compared to a simple design, especially in real-life industries.

2.0 Tech-based Work-in-Progress (Software Prototype)

Test	Key Functionalities	Test Descriptions/Scenarios	Expected Outcome	Status (Pass/Fail)
01	Set up the initial game board (including randomised positioning for dragon cards)	Chit cards Display	All 16 chit cards with different creatures are displayed on the screen as expected.	Pass
02		The initial state of chit-cards	When starting the game, all chit cards must show their back images.	Pass
03		Chit cards with varying numbers of animals on them.	All chit cards must have 1,2, or 3 animals for each different type of chit cards.	Pass
04		Chit cards must be shuffled.	The position of the chit cards is randomised when starting a new game.	Pass
05		Number of Caves	There must be 4 caves in the game board.	Pass
06		Position of Tokens	The tokens must be in their corresponding animal cave.	Pass
07		Volcano Cards Arrangement	There should be 24 volcano cards with 6 spiders, 6 salamanders, 6 baby dragons, and 6 bats.	Pass
08	Flipping of chit cards	Flipping Chit Cards	Players can hover their mouse to randomly flip dragon cards.	Pass
09		Animals Display on Chit Cards	The animal picture on the flipped chit cards is visible and aligned properly.	Pass

3.0 Executable Deliverable

3.1 Pre-conditions

Windows is the main platform for the executable file. Before opening and running the file, you would like to have Java SE Development Kit 18.0.2 and OpenJDK-18 (Oracle OpenJDK version 18.0.2) installed on your machine.

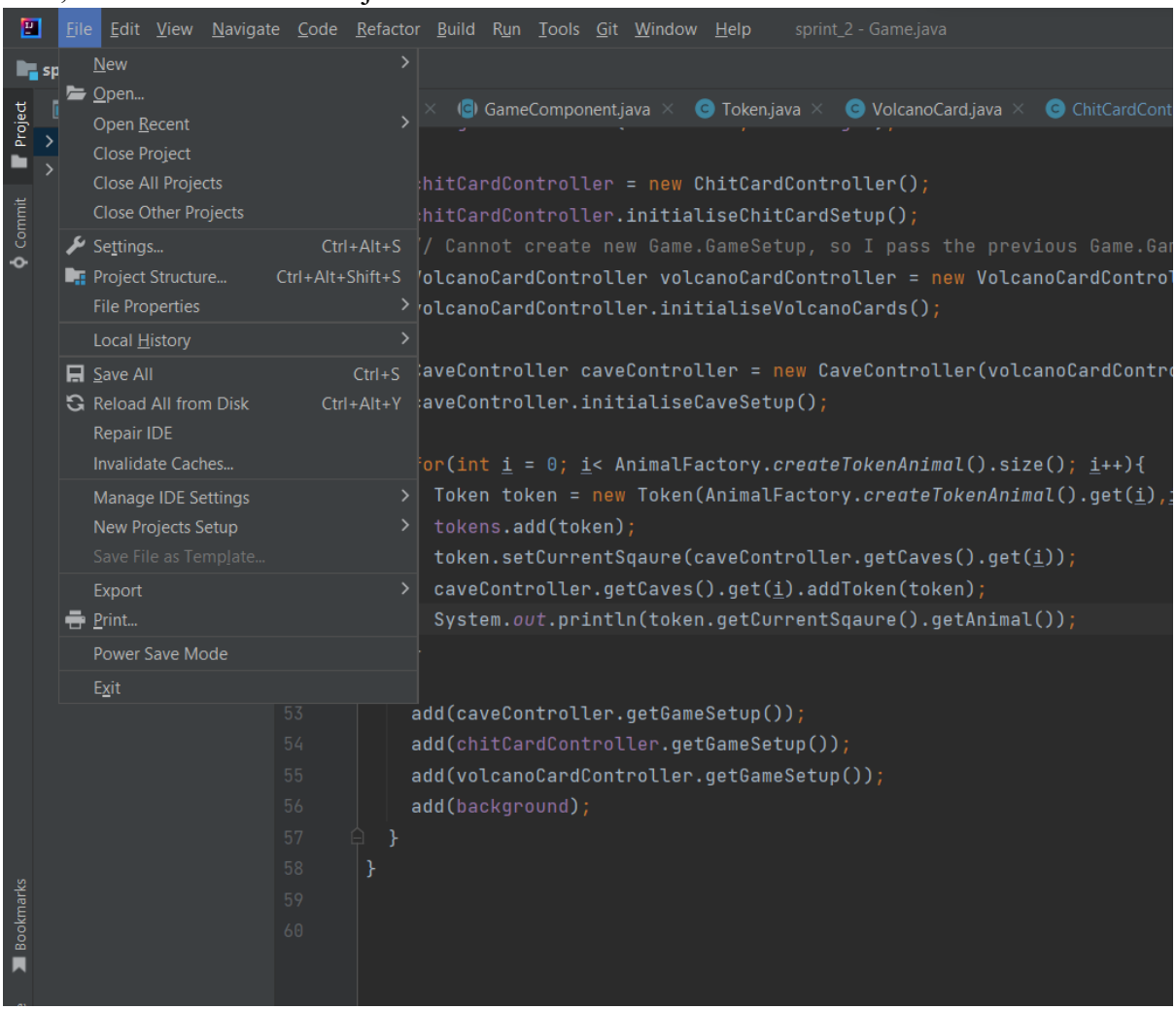
Here is the website link for installation:

https://www.oracle.com/java/technologies/javase/jdk18-archive-downloads.html

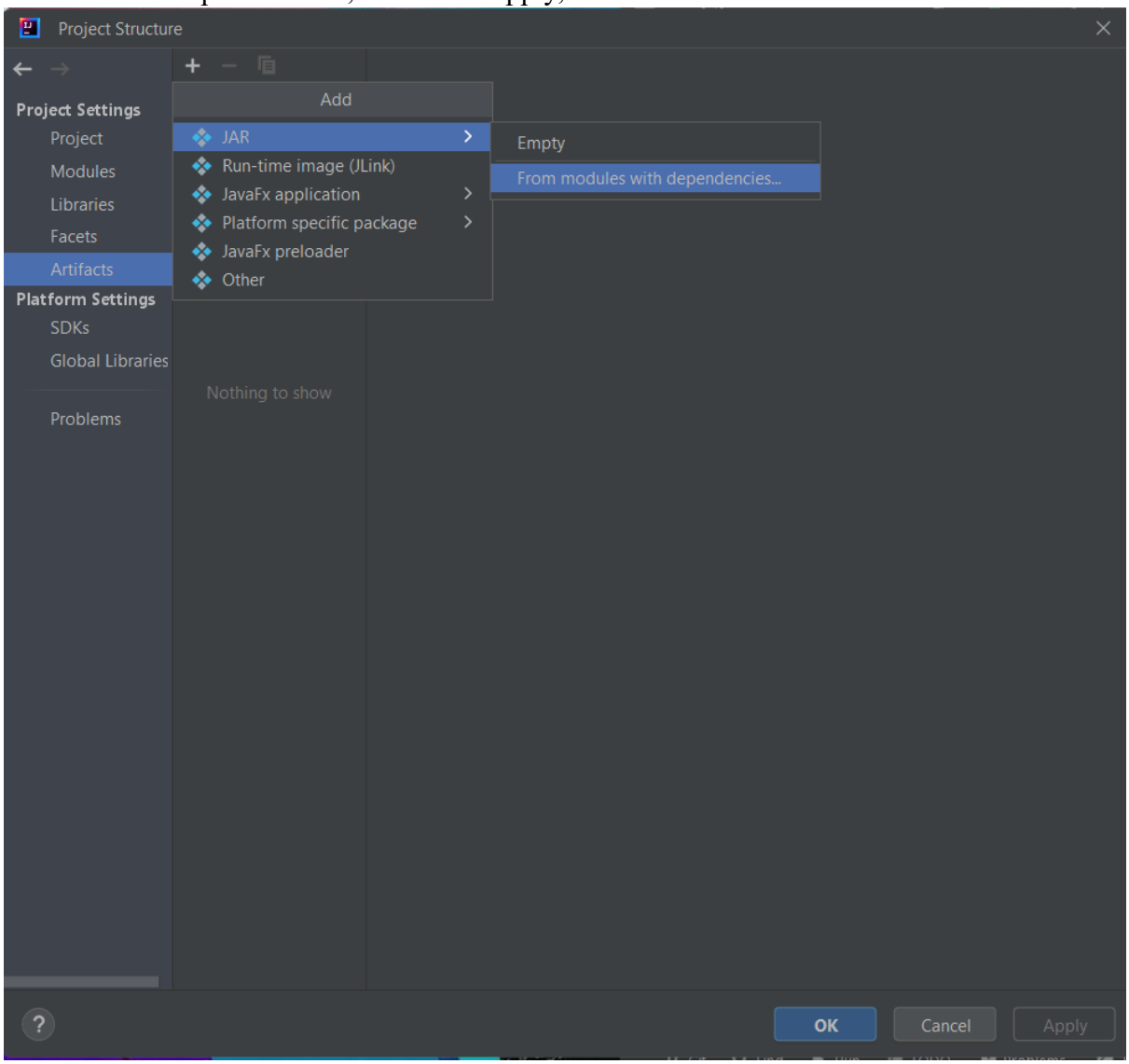
3.2 Building the executable file

I am using IntellijIDEA with my project, here are the build steps:

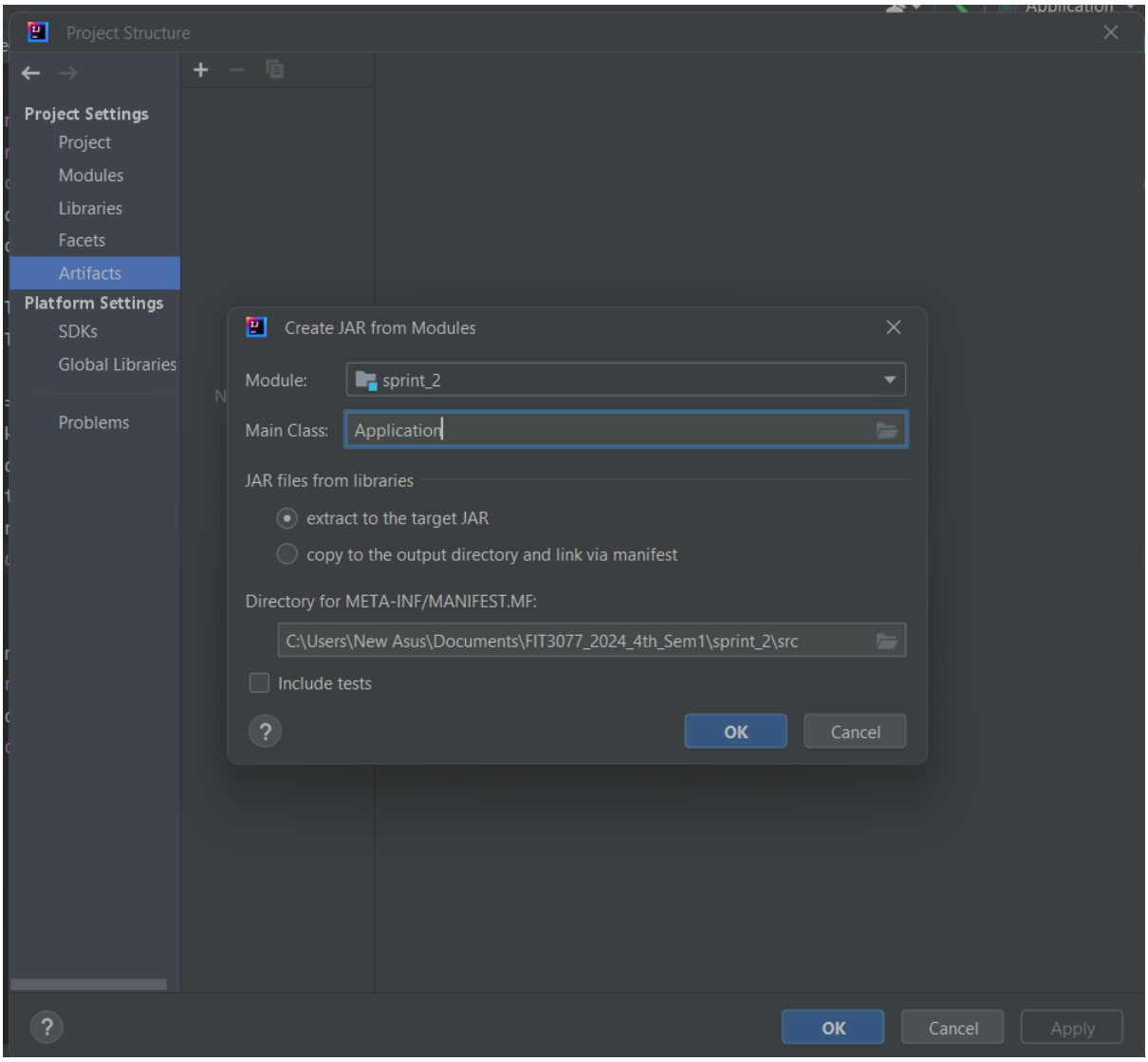
1. First, click on the File -> Project Structure…



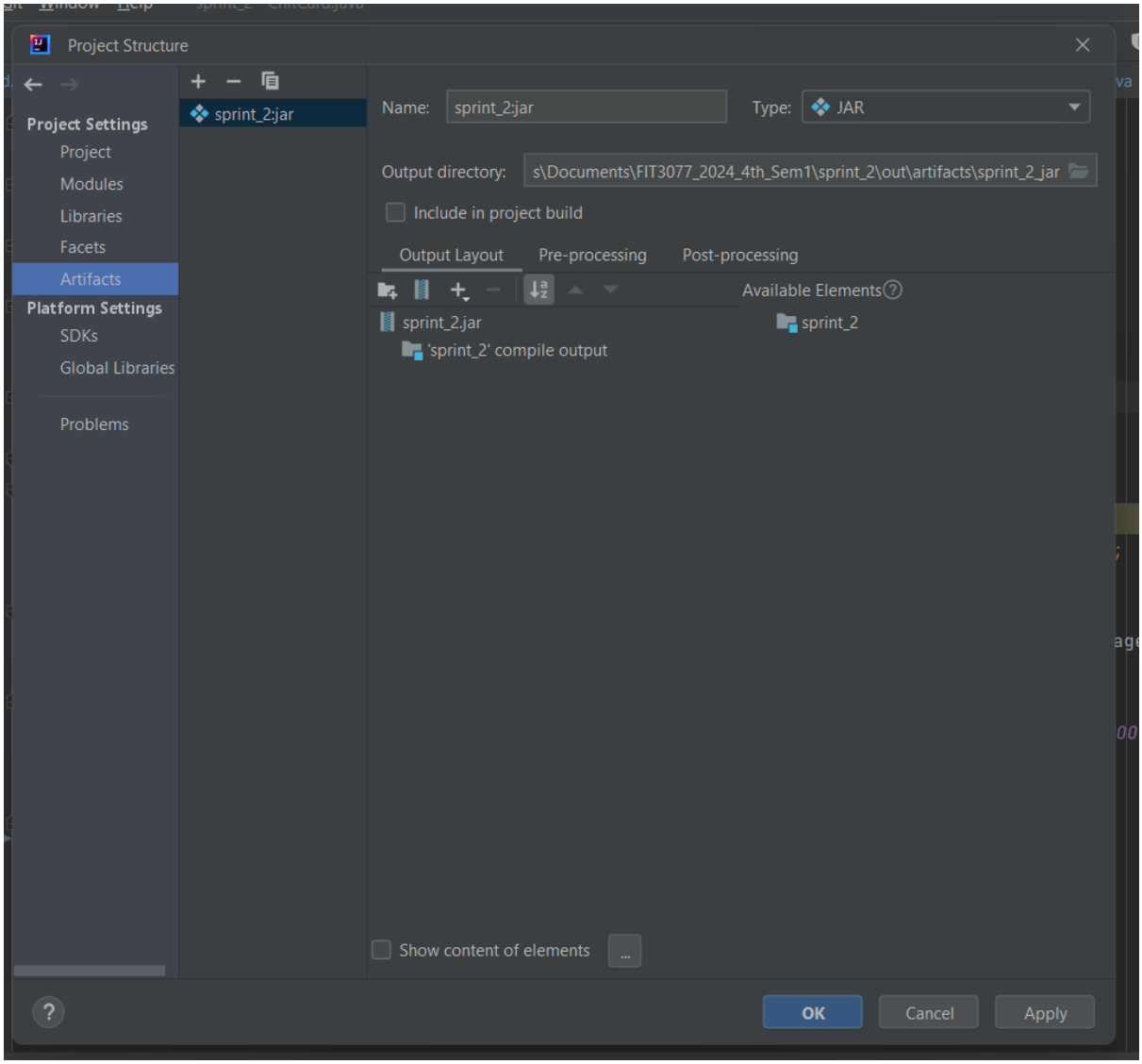
2. Next, navigate from “Artifacts” -> click on “+” signal -> select “JAR” -> “From modules with dependencies”, then click Apply, then OK.



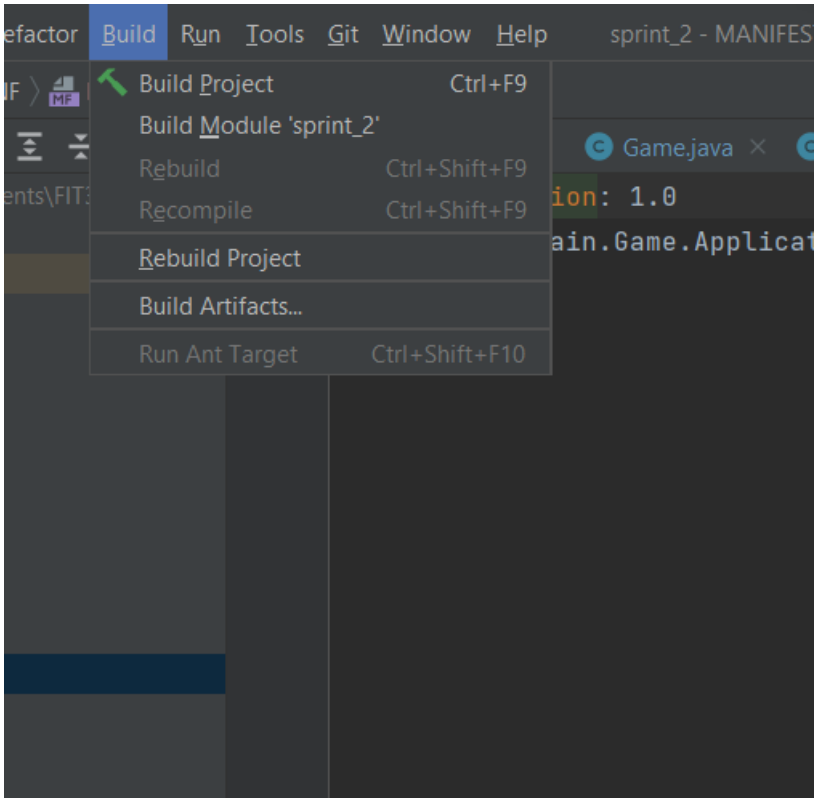
3. Choose the main class, and click OK.



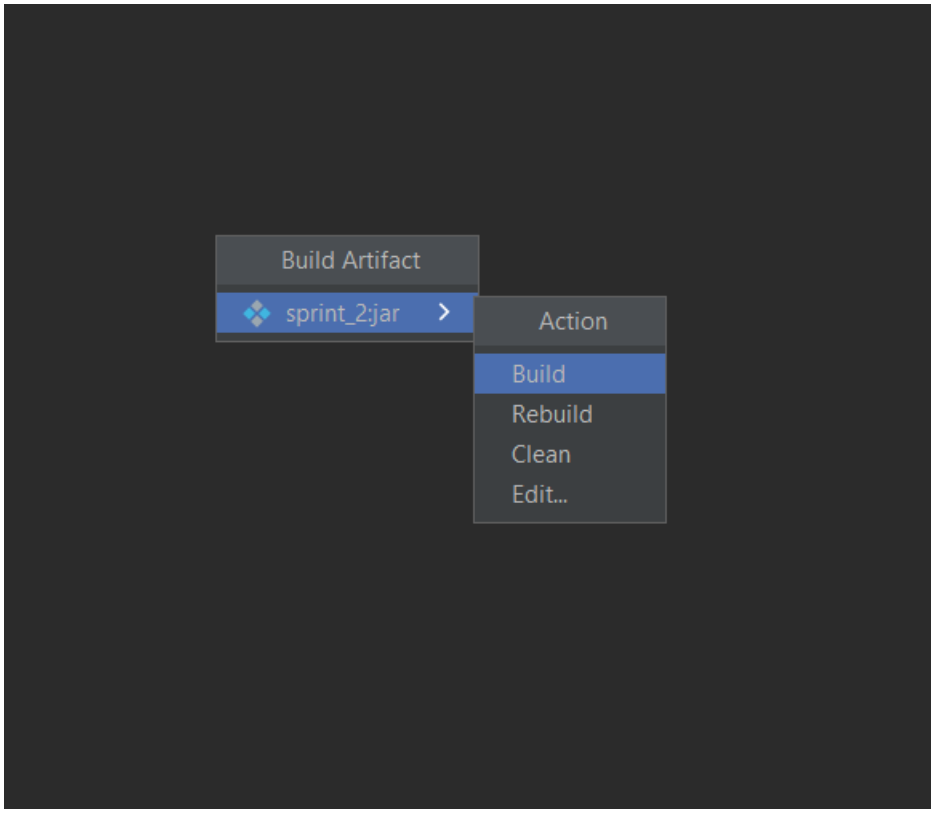
4. Navigate to Apply -> OK.



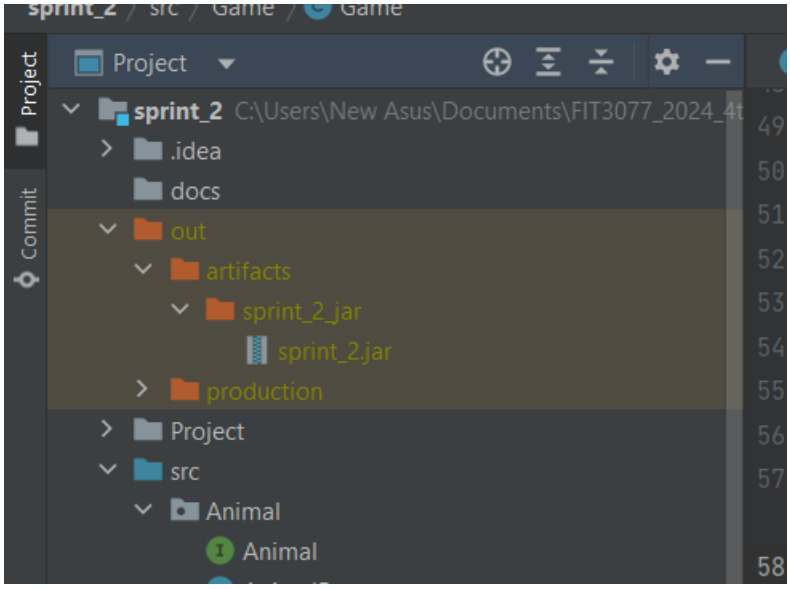
5. Next, click on “Build” -> “Build Artifacts”.



6. Click on “Build”.



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



3.3 Running the executable file

1. Download the sprint_2.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 sprint_2.jar

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

C:\Users\New Asus\Desktop>java -jar sprint_2.jar
```

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