FIT3077 Software Engineering: Architecture and Design
Sprint 1

Team Information

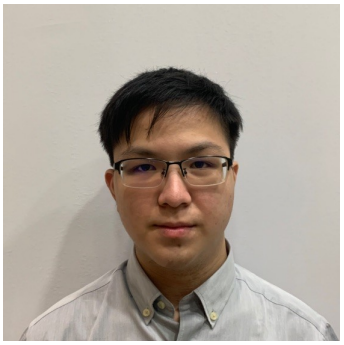
Team Name : Everything

Team photo

Team Membership

Naavin Ravinthran
Technical Strengths:
Familiar with C++ Stack (C++, CMake, SFML), Cybersecurity
Contact Details: nrav0005@student.monash.edu
Fun fact:
I pissed off an FBI representative before in real life.

Chan Jia Zheng
Contact Details : jcha0170@student.monash.edu
Technical Strengths :
OOP, C++ experience, data structure & algorithms.
Fun Fact:
I am allergic to abalone.



Name : Low Jun Jie
Contact Details : jlow0030@student.monash.edu
Technical Strengths :
Have experience in Game Development; Java.
Fun Fact :
Has Asthma due to inhaling concrete powder everyday when young.

Team Schedule

Weekly meeting on Friday & Sunday 10pm-12am. Every week, we will have two 2-hour meetings on assignments. The meeting duration will be extended or additional meetings will be conducted based on demand. The meetings are done using Discord.

All team members contribute to all deliverables. All work is distributed evenly among the team members because all work is done during the meeting.

Technology Stack and Justification

Our main Programming Language of choice is **C++**.
C++ is a powerful and widely used programming language that is ideal for developing complex applications that require high performance, efficiency, and control over system resources. It is also a language that is strongly based on object-oriented programming (OOP) principles, including encapsulation, inheritance, and polymorphism, making it an excellent choice for projects aimed at learning OOP. One extra thought is that all of our team members have at least some experience in **C++,** with the majority of the members being proficient at Modern **C++** and its development environment and the tools associated such as **CMake**.
Other discussed Languages to use are **Java** and **Python**.
While **Java** is a popular contender to use as all team members have taken **Java** as a unit before, the build system and the development environment still remains largely unfamiliar to most members of the team.
For **Python**, all members are proficient in **Python**, but it is not as suitable as the Object Oriented Principles are loosely maintained as per the specifications of **Python**, it may be easier to make the mistake of violating Object Oriented Principles using **Python**.

We will be using **SFML** library for providing an easy interface to graphics and audio.
SFML library is a popular and well-supported library for developing multimedia applications, including games, with **C++**. **SFML** provides a set of easy-to-use interfaces for graphics, audio, and networking, allowing us to focus on the game development and design instead of having to deal with low level graphics and audio handling, which can save a lot of time for actually developing the game. It also has a strong community and an abundance of examples, tutorials, and resources for learning how to use it effectively.
For the other 2 languages also discussed, popular libraries for handling graphics and audio include **LibGDX** and **JavaFX** for Java, and **PyGame** for Python.
Majority of the members are not familiar with **JavaFX** nor **LibGDX**, in addition, the former may have exporting issues to Linux Operating Systems, which one of the members experienced before as they use a Linux Driver; while the latter is slightly complex, it will take time to learn about the library and hence is not ideal for the project when a short time constraint is imposed.
As for **PyGame**, **PyGame** does not natively handle 3D, which may be one of the features under consideration, it may become a restriction if the library is used and we will have to develop the game from scratch. **PyGame** also has trouble exporting to Linux Operating Systems, which will be a problem to us as we ideally would want the game to be cross-platform compatible.

After consideration of all our team members' technical strengths, we will be using **C++** with **SFML** for this project. For development convenience, the use of tools such as **CMake** may be introduced during the development.

User Stories

No.	As a (Role)	I want to (Desirable Action)	So that (Benefit)
1	Game Player	move my token to adjacent points after placing all tokens	I can form 3 tokens in a row.
2	Game Player	move my token to adjacent points after placing all tokens	I can avoid the opponent to form 3 tokens in a row
3	Game Player	move my token to any vacant points when I have 3 tokens left	I can form 3 tokens in a row.
4	Game Player	move my token to any vacant points when I have 3 tokens left	I can avoid the opponent to form 3 tokens in a row
5	Game Player	move my token to any vacant points when I have 3 tokens left	The opponent doesn't have legal moves.
6	Game Player	place a token on a vacant point when I have token left	I can form 3 tokens in a row.
7	Game Player	place a token on a vacant point when I have token left	I can finish placing 9 tokens before I start moving the placed token
8	Game Player	Undo moves	I can undo my mistakes
9	Game Player	Save current game state to a file	I can restore it in the future
10	Game Player	Name the file of the saved game state	I can use a meaningful filename.
11	Game Player	Load a saved game state.	Resume previous progress.
12	Game Player	Capture an opponent's token not in a mill when I form a mill unless the opponent only has mills	I can reduce the number of tokens the opponent has on the board to win.
13	Game Board	Ensure no illegal moves are made	A fair game can be played
14	Game Observer	Determine win condition	So that players know when they have won and the game ended
15	Game Observer	Detects a mill	So that the gameboard knows there is a mill and let the player remove the token.
16	Game Board	Display the whose round it is.	The players know who should be playing in the current round.
17	Game Board	Display the player's current action	The players know what he should do in the current round.

18	Game Application	Let the the player to start the match	The game can be started.
19	Game Application	Let the player to restore a game state	The previous saved game can be continued.
20	Game Application	Let the player to quit the game	The game shuts down gracefully.
21	Game Application	Let the player to save the game state to a file	The players can continue the game next time.
22	Game Application	Display all the saved game state	The player can select which saved game to resume.
23	Game Application	Open the saved game correctly	The players can get the exact game state they were previously playing.
24	Game Application	Display the layout correctly	The use experience won't be correctly
25	Game Application	Switch between the scenes correctly	The user can navigate between scenes correctly.

Basic Architecture

The **MainApplication** class is responsible for swapping between windows in our game. The **MainApplication** class will use the **Enum Scene** to determine which scene should be rendered. The **UI** will return the next scene that needed to be rendered to the MainApplication when the user made an interaction. In the program, **UI** is a class for objects that are clickable. Token is clickable, GameBoard has UIs because it has points that are clickable. The **GameBoard** has **UI** as we see vacant points as clickable objects.

In the initial design, we have a **Player** class in our program. After careful discussion, we found that the **Player** class doesn't need to store anything and thus it doesn't have any important behaviour. Therefore, it is needless for us to create a simple class for **Player**. As a result, the **Player** appears as **Enum** in our program, it is used by the **Command** classes to keep track of which player made the command.

For the **EncoderDecoder** class, we will use this to encode and decode the game state between **GameMomento** and text. This class can be accessed by the whole game application (in different classes) and it doesn't need to store information nor to be modified. We create a class to encode and decode the game state due to *Single Responsibility Principle*, the other class should be carefree of the coding and encoding programs. It also creates abstraction to other classes because in the future the encoding and decoding algorithms and text format might change, the other classes can use the same code although changes have been made. In the initial design, the **MainApplication** is responsible for this feature because this method can be used across multiple scenes in **MainApplication**. However, we take in consideration that multiple algorithms might be used, this feature is big enough to have its own class.

In the initial stage of the design, the **GameBoard** is responsible to observe that game state of the game. We are using the *Observer Design Pattern* to observe the game state. This is because we do not want the game board to handle too many tasks, we want to follow the *Single Responsibility Principle*. In our opinion, the gameboard should focus on the tokens and execution of the commands by the players. In this case, there are only 2 observers made by us and the game board maintains them, the **GameBoard** is the subject. At the end of every round, the **GameBoard** will pass the grid information to the observers so that it can determine what command should be used in the next round of the game or end the game. In the initial design, the **GameBoard** is observing the states. In the initial stage, it is fine for **GameBoard** to observe the state, however the responsibilities of the **GameBoard** increases when the number of features increases. The **GameBoard** became a god class. Therefore this design is deprecated.

The **GameScene** will decide the next interaction moves (which command should be made by which player) based on the feedback from observers, the feedback is kept by. The feedback from observers will affect the Enum TokenState in Token for rendering purposes. For example in our program, the tokens will wear sunglasses when a player is removing an opponent's token in the current round. The same design idea goes to Enum GameState in GameState.

All visible elements will implement the interface **Renderable**. All elements in the current screen will render themselves when the screen refreshes.

For the functionality of *Saving* and *Restoring* games, we can save a ‘snapshot’ of the current game state, to be used for restoring later. The **GameBoard** holding the information of the current game state is where we would be able to create a data structure storing all the important information about a match, and also where we would want to load a game from. Thus, the GameBoard has to be able to perform *Save* and *Restore* to modify its own state to the saved data. We would abstract this mechanic out, where the game data contains all the history of Commands, elapsed time etc. and they are stored in the form of **GameMemento**. To use this **GameMemento**, we would implement *Save* and *Restore*, they are an interface abstracted out called Originator. This design partially follows the *Memento Pattern*, where since we currently do not have plans for restoring to a past saved Memento, we do not have the need for a *CareTaker* as described by the *Memento Pattern*, but it will provide us flexibility of keeping a collection of snapshots of a particular game if it is required.