FIT3077 Software Engineering: Architecture and Design

Sprint Three Submission

CL_Monday06pm_Team988

Group Report

Group Members:	**Email Address:**
Sheena Quah Xin Yee	squa0013@student.monash.edu
Samantha Oh Jia-Jia	sohh0008@student.monash.edu
Racheal Lim	rlim0050@student.monash.edu
Ng Jeh Guan	jngg0104@student.monash.edu

Table of Contents

1.0 Assessment Criteria

1.1 Functional Completeness & Correctness

1.1.1 Feature Coverage Ratio

The total number of features needed includes five fundamental game functions: establishing the initial game board (where the dragon cards are positioned at random), flipping the cards, moving the dragon tokens according to their current location and the most recent flipped card, alternating player turns, and winning the game.

Most prototypes receive a **functional completeness score of 40% (2/5 * 100%)**. A better functional completeness score of **60% (3/5 * 100%) is achieved by Racheal's** implementation, which stands out due to the incorporation of three of the five needed features.

1.1.2 Function Point Analysis

1.1.2.1 Samantha

Based on Samantha's class diagram, locateTokens() that were not in the planned diagram. Function points are assigned to each feature based on their determined complexity, which provides a quantitative measure of the effort and resource allocation needed for implementation. There are three categories for the complexity levels: low (3 points), medium (5 points), and high (7 points). Taking into account the previously determined difficulties, the precise assignments for every function are as follows:

- showWin() : 5 function points [medium]
- winningSituation() : 7 function points [high]
- displayHabitats() : 5 function points [medium]
- generatePlayers() : 4 function points [low-medium]
- resetGame() : 6 function points [medium-high]

The sum of the points for each intended functionality—5 for showWin, 7 for winningSituation, 5 for displayHabitats, 4 for generatePlayers, and 6 for resetGame—amounts to 27 total planned function points. On the other hand, including the extra locateTokens() method—which was not part of the initial design but was added to the project—the total number of **implemented function points comes to 31**.

The function point utilization is determined. This is found using the formula: ***Function Point Utilization = (Total Planned Function Points/Implemented Function Points for Planned Features)×100%***

With all planned features for the functionality implemented, the utilization is 27/27 x 100% = 100%. This outcome shows that the features that have been implemented and those that have been planned align perfectly, indicating that Samantha has successfully carried out the project's original goal.

1.1.2.2 Racheal

Similar to the analysis done for Samantha's project, we will evaluate the function points for Racheal's game development project based on the functions implemented. Assigned Function Points for Racheal's Implemented Functions:

startGame() : 5 function points [medium]
resetPlayer() : 3 function points [low]
initialize() : 7 function points [high]
displayShuffledDeck() : 5 function points [medium]
arrangeAnimalsInCircle() : 5 function points [medium]
shuffleAndDisplayAnimals(): 7 function points [high]

arrangeAnimalTokens() : 5 function points [medium]
intializeTokenViews() : 5 function points [medium]
initializeImageView() : 3 function points [low]
flipCard(int) : 5 function points [medium]
processCardEffect(Card) : 7 function points [high]
updateGameBoard() : 5 function points [medium]
updateCurrentPlayerDisplay() : 4 function points [low]

Total Planned Function Points (based on class diagram functionalities planned to implement):

- startGame(): 5
- flipCard(): 5
- checkForWin(): 7
- nextPlayer(): 5
- getPlayer(): 3
- Total Planned: 25 points
- Total Implemented Function Points:

Sum of points for implemented functions (based on the assigned points above): 71 points. With planned features of 25 function points and implemented function points of 25, the utilization is 100%. All intended features have been successfully implemented by Racheal, who has also included new elements that raise the game's difficulty and player involvement. The solution demonstrates strong project execution and creative improvements by not only meeting but exceeding the intended scope.

1.1.2.3 Sheena

Assigned Function Points for Sheena's Implemented Functions:

playGame(): 5 function points [medium] initialize(): 7 function points [high] initializeTokenViews(): 5 function points [medium] getPlayerTokens(): 3 function points [low] setPlayers(): 3 function points [low] initiliazeDeckImageView(): 5 function points [medium]	initializeShuffledDeck(): 5 function points [medium] displayShuffledDeck(): 5 function points [medium] flipCard(int): 5 function points [medium] displayVolcanoCards(): 5 function points [medium] arrangeCaves(): 5 function points [medium] nextPlayer(): 5 function points [medium] updateCurrentPlayerLabel(): 3 function points [low]

Sum of points for implemented functions (based on the assigned points above): 61 points. Total Planned Function Points (based on class diagram functionalities planned to implement):

- playGame(): 5
- setCurrentPlayer(): 3
- flipCards(): 5
- nextPlayer(): 5
- hasWon(): 7

Sum of points for planned functions (based on the assigned points above): 25 points. All planned features have been implemented except for hasWon() as it is a functionality under Samantha's scope of responsibility. Instead, Sheena implemented the nextPlayer() function which was not part of the class diagram and has a slightly lower function point compared to the hasWon() function. This gives a 92% function point utilization.

1.1.2.4 Jeh Guan

Assigned Function Points for Jeh Guan's Implemented Functions:

- resetGame(): 5 function points [medium]
- initialize(): 5 function points [medium]
- setUpMovementCard(): 5 function points [medium]
- setUpHabitatSymbols(): 5 function points [medium]
- flipMovementCard(): 5 function points [medium]
- coverAllMovementCard(): 5 function points [medium]
- displayHabitatStatus(): 5 function points [medium]

Planned Function Points (each feature is of medium complexity and worth 5 function points) *excludes getter and setters:

- randomiseMovementCard()
- setUpHabitats()
- randomiseHabitats()
- setUpCave()
- setUpAnimals()
- checkMovementCard()
- isCurrentAnimalTurnEnded ()
- isAnimalWin ()

Total Function Points is 35 points (implemented) + 40 points (planned) = 75 points. The function point utilization percentage, based on the adjusted function points, is approximately 46.67%. Regarding the implementation of isAnimalWin and other unimplemented features, it's normal as it's out of the scope of Jeh Guan's task in sprint 2.

1.2 Functional Appropriateness

1.2.1 Function Utilization

The Function Utilisation metrics evaluates how often and to what extent end users use a system's features and functions. This measure is especially crucial for determining which aspects of a system are most useful to consumers, which could use some tweaking, and which might be unnecessary.

1.2.1.1 Samantha

By looking at Samantha's sequence diagram, the checkifWin() function is called every time a player makes a movement. This results in a high usage frequency, suggesting that the function is critical to the gameplay experience. It also indicates that the function needs to be highly optimized to handle being called frequently without affecting game performance. For example, if the logs from a testing session indicate that 80 player moves occurred in an hour, then that means that the checkIfWin() method was called 80 times during that testing session. As shown in the logs when code is runned, usage frequency of the checkIfWin() function equals the number of player movements.

1.2.1.2 Racheal

The function processCardEffect(Card card) is crucial to the workings of the game. The duties assigned to this function include changing various aspects of gameplay, player placements, and the game state based on the card qualities. Because it is activated each time a card is flipped or utilized, it is particularly often used and an essential part of the game. Because of its critical function, processCardEffect(Card card) must be optimized for efficiency in order to prevent any possible lag or performance problems when playing. The sequence diagram designed by Racheal indicates that getPosition() is used every time processCardEffect(Card card) iterates, indicating that verifying the state changes brought about by card effects is an important function of getPosition(). Its frequent application highlights how crucial it is to preserve the integrity and coherence of the game's state at the beginning of each turn.

1.2.1.3 Sheena

According to Sheena's sequential diagram, the nextPlayer() function gets called every time a change of turn occurs. A change of turn occurs whenever the animal of the card that the player flipped does not the animal of the habitat that the player is in, when a pirate card is flipped, or if the animal on the card flipped does not match the player's cave type when the player has yet to set foot on the volcano cards. This has a medium to high usage frequency as it depends on the player interaction with the game. For example, the usage frequency of this function will be lower when the majority of the players are able to remember the position of the cards that would benefit them, as compared to when players are unable to remember the card positions which results in a change of turn every time a card is flipped. This is one of the essential functions in the game to maintain game balance, ensuring that each player gets to play the game fairly. Therefore, it is critical that this function is optimised to reduce game latency and improve game performance.

1.2.1.4 Jeh Guan

Since it starts each player's turn, the flipMovementCard(int indexOfMovementCard) function is essential to

the game. Each player must flip a movement card to proceed, making this function central to gameplay. Its critical role is highlighted by the frequency with which it is used; for example, if players can flip the card 16 times in 100 turns, the function is called 1600 times.
The player's next moves are determined by this function, which displays the card at the designated index. It starts important game operations like choosing movement options and changing game states. It can also start other important operations like processCardEffect(Card card) or nextPlayer(). Owing to its importance, `flipMovementCard()` needs to be as efficient as possible in order to prevent latency problems and guarantee fluid gameplay.
Maintaining flipMovementCard() integrity and functionality is crucial to avoiding game flow interruptions and preserving the fun for players. Its robustness directly impacts the game's quality, making it indispensable to core mechanics.

1.3 Appropriateness Recognizability

1.3.1 Time on Task

1.3.1.1 Samantha

We can follow a comprehensive procedure to collect and analyse pertinent data in order to effectively evaluate the "time-on-Task" metric for Samantha's implemented software features.

Steps to Evaluate Time-on-Task for Samantha's Features

Step 1: Define the Tasks

- Task 1: Setting up the initial game board with a shuffled deck of cards.
- Task 2: Placing shuffled animal cards on the game map.
- Task 3: Executing the winning situation in the Fiery Dragon game.

Step 2: Gathering Information

Gather time information for every job. This may be accomplished by watching as several players set up the game for the first time. To measure how long it takes to complete a task—that is, to get the game board and map ready for play—you may use a timer to measure the time. In a similar vein, note how long it takes a player to successfully fulfil a winning condition from the moment it first becomes available.

Step 3: Calculate Average Time-on-Task

For Task 1 & 2 (Automated): Measure the time from clicking the "Start" button to when the board and cards are fully set up and ready for gameplay. This measures the efficiency of the automation rather than user interaction. **For Task 3 (User-Initiated):** Record the time from clicking the "Next" button to the display and recognition of the winning situation.

Determine the average time needed for each job throughout the course of all observed sessions:

- Task 1: Average Time on Task = 3 seconds
- Task 2: Average Time on Task = 3 seconds
- Task 3: Average Time = 5 seconds

Step 4: Contextual Analysis

While the automation is well-received, there appears to be some confusion regarding Task 3, which is initiated by the "Next" button. Out of 10 testers, 2 did not immediately understand the purpose of the "Next" button and struggled with directing themselves to this task, which involves executing the winning situation

1.3.1.2 Racheal

We use an organized method similar to Samantha's to assess the "Time-on-Task" metric for Racheal's software features, which include assembling the initial game board with a shuffled deck of cards, positioning shuffled animal cards on the game map, tossing cards, and moving animal tokens in response to flipped cards and their locations on the game map. This evaluation will provide light on the effectiveness with which these features allow users to complete their duties and the program's functional suitability.

- **Task 1**: The game board took an average of 2 seconds to put up completely
- **Task 2:** After the initial setup, it took an average of 2 seconds to place the animal cards.
- **Task 3:** Including the time it took for players to make up their mind, players usually took 5 seconds to choose and flip a card.
- **Task 4:** No matter the distance traveled and the intricacy of the move as determined by the card effects, moving animal tokens took an average of just around 2 seconds.

Due to their automation, tasks 1 and 2 exhibit great levels of efficiency and fast setup times. This automation contributes to the fast and captivating game speed. Tasks 3 and 4 had somewhat longer timeframes and required direct player participation. This makes sense because these activities call for the user to engage with the game's UI and make decisions.

1.3.1.2 Sheena

Key tasks in the program include:

- **Task 1**: setting up the players in the game
- **Task 2**: setting up the game board with volcano cards
- **Task 3**: arranging the shuffled game cards on the game board
- **Task 4**: player flipping a card
- **Task 5**: ending a player's turn and pass the turn to the next player

Average Time-on-Task

- **Task 1**: 1 second
- **Task 2**: 2 seconds
- **Task 3**: 2 seconds
- **Task 4**: 5 seconds
- **Task 5**: 3 seconds

Sheena's program executed the setting up the initial game state (tasks 1-3) efficiently, with an average of 2 seconds. On the other hand, tasks 4 and 5 required user interaction with the game. Similar to Racheal's program, players would take 5 seconds on average to decide which card to flip. As for the change of turns, Sheena programmed this functionality in such a way that there is a short delay between after a card is flipped and when the change of turn is executed. This is because players need to be notified on what caused the change of turn before changing the game state, instead of immediately flipping the cards back over after a card has been unfolded.

1.3.1.2 Jeh Guan

Tasks:

1. Setting up the initial game board.
2. Enabling the cards to be flipped.

Task 1: Setting up the initial game board

- Efficiency: The automated setup takes an average of 2 seconds, ensuring a quick start to the game. This minimizes downtime and enhances user experience by allowing players to begin playing promptly.

Task 2: Enabling the cards to be flipped

- User Interaction: This task, involving player interaction, takes an average of 3 seconds per card flip. The system's quick response time ensures smooth gameplay, balancing responsiveness with the need for player decision-making.

Overall Evaluation:

Jeh Guan's implementation balances efficient automation with responsive user interaction. The swift setup (2 seconds) and optimized card-flipping process (3 seconds) enhance the gaming experience by providing a quick start and engaging, seamless gameplay, aligning with functional appropriateness in ISO/IEC 25010.

1.4 Modifiability

1.4.1 Dependency measures

1.4.1.1 Samantha

To assess and calculate the dependency measure in Samantha's class diagram, we can consider two types of relationships highlighted: Association and Aggregation; whereas conclude its measure with the formula Dependency Measure (Class) =Number of Classes Directly Depended Upon.

Assigned Functional Dependency for Samantha's Classes:

- GameController : GameMap, CardsOnMap [2]
- GameMap : Animal, Cave, Habitat [3]
- CardsOnMap : Card [1]
- Card (Abstract) : CardType [1]
- AnimalCard & PirateCard : Inherited from Card [1]
- Animal : AnimalType [1]
- Cave : Animal, Position [2]
- Habitat : Animal, Position [2]
- Position : *only interacts with other classes [0]

Classes having more dependents (like GameMap) in the above class diagram show a higher degree of coupling, which might make adjustments more difficult because of potential ripple effects. On the other hand, classes with few or no dependencies, like Position, are simpler to update and maintain. Reducing these dependencies is a goal of good software design in order to increase modifiability. Reducing direct dependencies between concrete classes may be accomplished by using interfaces, abstract classes, or design patterns like Observer or Mediator.

1.4.1.2 Racheal

The GameController's moderate to high reliance in the Racheal's game's class diagram is shown by its aggregation with the Player, which is rated as a moderate dependency at two locations. Shown in Racheal's designed class diagram, the GameController has weak dependencies with the GameMap and other related entities, each rated at one point. As a consequence, a total dependence score is produced that takes into account all of these linkages and shows that, even though the GameController is essential to the game's functioning, it still has a flexible relationship with its entities. This adaptability is essential for a more flexible gameplay experience in a dynamic gaming environment where player interactions might change dramatically.

On the other hand, because of its compositional link with the Habitat, which is rated as a strong reliance and given three points, the GameMap shows a higher degree of dependency. Additional relationships with components such as Animal and Card are regarded as weak dependencies, adding to the overall dependency score of the GameMap. This high dependence score indicates a close-knit relationship in which modifications to the GameMap would need matching changes to the Habitat class. Although this intimate relationship is essential to preserving the game's structural integrity, it may provide difficulties for any future gameplay expansions or alterations, thereby limiting the game's potential for flexibility.

1.4.1.3 Sheena

Assigned Functional Dependency for Sheena's Classes (according to the class diagram):

Class	**Dependency**	**Number of Dependencies**
GameController	Player, Card(abstract)	2
GameMap	VolcanoCard, AnimalCave	2

BabyDragon, Fish, Pufferfish, Octopus	Inherit from Animal(abstract)	1
AnimalCard, PirateCard	Inherit from Card(abstract)	1
Player	Token	1
VolcanoCard	Habitat	1

The aggregation relationship between GameController and Player in the class diagram shows a medium-high dependency of GameController on Player. The composite relationship between GameMap and VolcanoCard implies a tightly coupled relationship between them. Making changes to classes with higher (or more) dependencies like so may slow down the development process as it may be hard to modify one class without affecting the other, which is a sign of reduced flexibility. Oppositely, Animal classes, Card classes, Player class, and VolcanoCard class have weaker dependencies and lesser number of dependencies on other classes. These classes hence offer more flexibility to the program for future changes.

1.4.1.4 Jeh Guan

Jeh Guan's class dependencies are as follows:

GameMap: Aggregates Animal, Habitat, Cave, MovementCard [4]
GameController: Aggregates GameMap [1]

The GameMap class has a high dependency, aggregating four classes: Animal, Habitat, Cave, and MovementCard. This tight coupling indicates that changes to these aggregated classes will likely affect GameMap, making modifications more complex and potentially causing ripple effects across the system. Conversely, the GameController class has a single aggregation with GameMap, resulting in a lower dependency measure of 1.

Classes like GameMap with higher dependencies reduce flexibility, complicating updates and maintenance. In contrast, GameController's lower dependency measure suggests it is more flexible and easier to modify. Improving modifiability could involve decoupling these classes through design patterns like Observer or Mediator, reducing direct dependencies, and enhancing the system's overall flexibility and maintainability.

1.5 Maintainability

1.5.1 Coding Standards

1.5.1.1 Samantha

Samantha's implementation adheres to a structure and naming convention that are consistent, which is essential for code that is easy to understand and maintain. Nonetheless, several aspects of the coding standards have been found and require more improvement. Certain approaches carry out several tasks that might be better divided into more manageable, targeted tasks. For example, locateTokens() manages both location computations and user interface changes; these may be divided to improve modularity.

The majority of the source code lacks considerable use of case analysis, such as elaborate if-else chains or switches, which might make it more difficult to comprehend and maintain the code. A simple flow is suggested by the sparing use of sophisticated conditional logic, which is usually simpler to test and debug. Samantha's code does not explicitly show down-casting, which is good as it avoids potential runtime type errors and maintains type safety. The code does not appear to handle many errors. To handle problems or unexpected behaviour, error handling methods are essential, especially when working with UI components and external resources like picture files.

1.5.1.2 Racheal

By dividing various functionality into methods like startGame(), resetPlayer(), initializeDeck(), etc., Rachel's code exemplifies good modularity. Still, in terms of modularity, there's potential for development. For

example, startGame() manages several functions, including initialising the game logic and user interface. To make it easier to manage, it might be divided into smaller, more targeted procedures.
Racheal's code includes comments, which help to understand the purpose of methods and blocks of code. To make her writing easier to read and comprehend, Racheal keeps her control structure rather basic and uncomplicated. Still, there are several places where the readability may be strengthened. To improve clarity, the initializeDeck() method, for instance, may be refactored, particularly with reference to card generation and dealing.

Rachel uses object-oriented concepts like encapsulation to good use, as seen by the Card, AnimalCard, and PirateCard classes. Cards and the functions they are connected with are represented succinctly and clearly through the use of abstraction.

1.5.1.3 Sheena

All main functionalities including initialisation functions are coded separately in Sheena's program so that it does not overwhelm the playGame() function. This makes her code seem more tidy. Naming conventions throughout the code were also consistent, and there were comments explaining the logic of most of the functions and codelines, which makes it easier to understand the logic flow of the entire program. Despite this, some codeline like in the initialisation functions (i.e. setting up the board and generating cards) can include more detailed comments. Moreover, there were no extra variables in the code that were not used throughout the program, which also improved the overall readability of the program.

While the overall code does not demonstrate down-casting, there are some improvements to be made. For example, defining constants for static values such as image dimensions, radius, angles, etc. to avoid definition repetition throughout the code. Furthermore, repetitive code such as those for positioning game elements (tokens, cards, etc.) can be refactored into utility functions. The GameController class contains code that handles game logic and UI updates, which can be divided into separate classes to prevent the GameController from becoming a "god" class.

1.5.1.4 Jeh Guan

Jeh Guan demonstrates strong adherence to coding standards, fostering maintainability in his software implementation. Consistent naming conventions ensure readability, while detailed function logic comments aid understanding, aligning with best practices for maintainable code. Each function adheres to the Single Responsibility Principle (SRP), promoting clarity and facilitating future modifications. Jeh Guan's avoidance of overwhelming functions further supports code maintainability by preventing excessive complexity. However, the GameMap class's potential accumulation of game setup logic suggests a risk of becoming a "god class," which could hinder maintainability. To mitigate this, redistributing some setup logic to other classes is recommended. By decentralizing responsibilities, the codebase remains modular, facilitating easier maintenance and reducing the risk of bloated classes. Overall, Jeh Guan's commitment to coding standards lays a strong foundation for maintainable software development.

1.6 User Engagement

1.6.1 Clarity & Simplicity

1.6.1.1 Samantha

Tokens are arranged in a circular pattern around the game board of Samantha's implementation, which is centred to maximise area utilisation and facilitate players' viewing of every aspect of the game. The logical locations of the START/RESET and NEXT buttons (bottom left and right) and their alignment with traditional UI layouts can aid in intuitiveness. Players can immediately recognise different game aspects because to the animal iconography' distinctiveness and artistic consistency, which also contributes to the game's overall thematic cohesion. This is important for games where gameplay is affected by rapid identification.

1.6.1.2 Racheal

The inner card circles and game tokens are arranged in an understandable circular pattern that makes it simple to comprehend the structure of the game. The distinct separation between the inner circle (cards) and outer ring (tokens) makes it easier for players to tell apart various game components. In order to guide gaming, the

interface also shows the current player ("Fish's turn") and the activities they are supposed to take ("moves 2 steps forward"). There is less chance of miscommunication over whose turn it is and what has to be done because this information is plain and straightforward.

1.6.1.3 Sheena

All 16 animal cards are arranged in the middle of a game board in 4 rows, with 4 cards in each row. Animal tokens are arranged in a circular pattern on top of their corresponding caves which closely resembles the physical game itself. A label on the bottom left corner shows the current player-in-turn. When the player flips an animal card that results in ending their turn, there is a two second delay before covering all unfolded cards to present a clearer picture of the game state to the players.

1.6.1.4 Jeh Guan

Jeh Guan's design enhances user engagement through clear and simple UI elements. Tokens are arranged in a circular layout around the game board, maximizing space and ensuring all game elements are easily visible. The START/RESET button and display labels are placed at the bottom left and right corners, adhering to conventional UI layouts for intuitive interaction. Distinctive and artistically consistent animal icons enable quick recognition, contributing to thematic cohesion and aiding gameplay by ensuring players can easily identify different game elements.

The game structure is straightforward, with inner card circles and game tokens arranged in an easily comprehensible circular pattern. The clear separation between the inner circle (cards) and outer ring (tokens) helps players distinguish various game components. The interface prominently displays the current player ("Fish's turn") and their required actions ("moves 2 steps forward"), minimizing confusion about turns and actions, thereby enhancing engagement through clarity and simplicity.

Additionally, Jeh Guan's arrangement of 16 animal cards in a 4x4 grid at the center of the game board, with animal tokens placed circularly atop their corresponding caves, mirrors the physical game layout for familiarity. A label at the bottom left corner indicates the current player. This thoughtful design ensures clarity and effective user engagement.

2.0 Summary Review

2.1 Jeh Guan (Flipping card)

Jeh Guan's software prototype, focuses on the flipping card function, the well-organized code structure and effective use of comments enhancing Jeh Guan's code readability and comprehensibility. A metric that Jeh Guan could aim to optimize further is the proactive adjustment of the game's scope which can demonstrate the flexibility needed in software development when refining his prototype.

Considering these insights, for Jeh Guan's flipping card function, it's essential to prioritize readability, modularization, and adherence to user interface conventions. Refactoring the function to enhance modularity and clarity while ensuring it aligns with user expectations can improve overall user experience. Additionally, adopting consistent coding standards and incorporating informative comments can aid future maintenance and facilitate collaboration within the development team.

2.2 Racheal (Movement)

Racheal's software prototype showcases good function point utilization, with a total of 79 assigned function points for implemented functions against an initial plan of 25. In terms of software design, Racheal's code is well-organized into distinct functions, though there is potential for increasing modularity by further breaking down complex functions. The use of code comments makes code more understandable and gives a simple control structure, yet some of her functions, such initializeDeck(), may use further clarification. Classes like Card, AnimalCard, and PirateCard, which cleanly encapsulate game elements and their interactions, demonstrate the effective application of object-oriented concepts like encapsulation. The user interface has been carefully created to distinguish between different game aspects and offer clear instructions on how to play.

2.3 Sheena (Change turn)

Sheena's software prototype demonstrates a good overview of the functionality in her design. All intended features of the base game were implemented and working well, excluding those that were not under her scope of responsibility. Utilisation rate could be said to be 100%, considering the functionalities she was responsible for. Code comments and distinct functions produce a cleaner codebase and ease code comprehension. Some of her functions and variables to set up the game board were repetitive, and can be optimised by refactoring them into utility functions and final constants (created with the Java keyword "final"). UI updates were also made in a way such that the game state is presented clearly to players. Overall, Sheena's final prototype approach demonstrates a well-executed implementation of the intended features, with a clean and comprehensible codebase.

2.4 Samantha (Winning game)

Samantha's software prototype complies well with the functionality indicated in her class diagram, and it successfully implements intended features like creating player profiles, figuring out winning conditions, and showing habitats, among other things. The unplanned inclusion of the locateTokens() method shows a proactive adjustment and expansion of the game's scope, adding even more value. Each feature has been given a certain number of function points that correspond to a careful assessment of difficulty; this results in a planned utilization rate of 100% for all 27 function points.

The average time to execute a winning condition when the user initiates the job is 5 seconds, suggesting a somewhat more complicated procedure that is nonetheless completed within a respectable amount of time. Minor problems with user comprehension of the "Next" button's operation were noted, nevertheless, indicating the need for more explicit instructions or UI changes to improve intuitiveness. Overall, Samantha's prototype approach shows a strong base with lots of code flexibility, adherence to a consistent structure and naming convention that contributes to the code's overall maintainability.

3.0 Assessment Outcome

3.1 Elements to use for Sprint 3

Following considerable deliberation, we chose to base the game on Racheal's code as we found her approach to be the most comprehensive, as it included the essential features of moving tokens and flipping cards. It built a solid foundation for us to improve the code quality and add our other features on top of the code. To make it easier for players to move tokens around rather than having to manually calculate the position of the next token that would go on each round, Racheal built a list of positions to store the position of each habitat. Following game rules, Sheena and Jeh Guan included a Volcano class to store three random habitats and generated a game map by grouping 8 volcano cards. This addition helped structure the game more effectively. To further refine the code quality by following OOP principles, Samantha employed enumeration to represent animal types instead of creating an Animal class or using string to define their types. Since the different animal types did not require unique functionalities, creating separate classes for each animal would have been redundant. Enumerations simplified the code and preserved its readability without adding needless complexity.

3.2 New ideas to improve prototype

As the number of habitats is fixed, which is 24, each token has to move forward for 24 + 2 (step out and step into its cave) steps to win the game. By counting the number of steps taken by each token, we can track the progress of each token instead of checking its current position every round and avoid the token exceeding its cave, as it only has to traverse one round to reach its cave.

Going forward, as we have to follow the game rule that states we only shuffle the volcanic cards when we restart the game, the singleton design pattern is a useful method for "locking" the habitats in each volcano card. We only need to generate 24 habitats at the very beginning of the game and distribute them randomly among volcano cards. After that, there's no longer a need to generate a new volcano card or new habitat when

the game restarts, which we call the instance of the VolcanoList (Singleton) instead.

4.0 Object Oriented Design
4.1 Class-Responsibility-Collaboration (CRC) Card

GameMap	
Create habitats	Habitat, Volcano
Generate animal caves based on the types of animals	Cave
Provide access to habitats and animal caves	Habitat, Cave
Shuffle habitats	Volcano
Set starting location for all caves	Cave

GameMap:
The game map shows where habitats and animal caves are displayed. It initialized habitats and animal cavers, shuffled the list of habitats, and offered methods for accessing habitats and caves.

Player	
Control its token	AnimalToken
Move the token on the game board based on a given number of steps	AnimalToken, GameMap
Provide access to token and positions	AnimalToken
Reset the player's position to the initial position	Player

Player:
Represents a player in the game. It includes information about the player's animal token and position on the game board, allowing the player to move their token in a predetermined number of steps, and provides methods to access the player's position and animal token.

GameController	
Manage the entire game flow	GameMap
Handles player interaction with the game elements (i.e. animal tokens, game cards)	Player, AnimalToken, Card
Handle player turns	Player
Manage the game map and its game cards	GameMap, Card
Determine player’s winning situation	Player, Cave

GameController:
The GameController class is the core coordinator for game flow and logic, which is mainly responsible for

handling player interactions within the game and game state (i.e. changing player turns and determining winning situation), and managing the game map's elements.

Habitat	
Hold a list of animal habitats	AnimalType
Check to see if the habitat contains an animal	Habitat
Check to see if the habitat contains a specific type of animal	AnimalToken

Habitat:
Represents a habitat on the game map where animals stand. It stores a list of animal habitats with different animal types, and provides methods for retrieving the list of habitats and determining if the habitat contains a given type of animal, and allows for the shuffling of habitats inside the list.

Card	
Determines the number of steps to be taken by a player	AnimalCard, PirateCard
Represents the game cards in the game map	AnimalCard, PirateCard
Encapsulates the logic to apply movement based on the type of card flipped	Player, GameMap

Card:
The Card abstract class provides a framework for the two types of cards used in the game: AnimalCards and PirateCard. It incorporates the shared behaviour and attributes of these card types such as the number of steps to be taken by a player. It also introduces an interface method which implements the movement logic of players on the game map based on the card flipped.

AnimalToken	
Represents an animal token on the game map	AnimalType
Tracks the position of the token on the game map	Player
Records the total number of steps taken by the token	Player

AnimalToken:
The AnimalToken class represents the individual tokens used by players on the game map. It mainly records the token's state by tracking the total number of steps taken by a player during the game and its position on the game map.

4.2 Class Diagram