Dragonix

Fiery Dragons

Sprint Three

Version 1.0

Prepared by:
Brian Nge Jing Hong
Wee Jun Lin
Ahmad Hafiz Bin Zaini
Sineth Fernando

Dragonix

Contents

1. Review of Sprint 2 Tech-based Software Prototypes

1.1 Assessment Criteria

To ensure the success of our software project, it's essential to assess the solution approaches proposed by each team member. Using criteria aligned with the ISO/IEC 25010 quality model (see Figure 1.1), we evaluate the functional completeness, rationale, understandability, extensibility, code quality, and user interface aesthetics. Functional completeness and correctness will ensure all key functionalities are covered without errors, while rationale will assess the appropriateness of chosen solutions. Understandability will focus on clarity, extensibility on adaptability for future enhancements, and code quality on maintainability. Finally, user interface aesthetics will enhance engagement. Through this evaluation, we were able to make informed decisions and foster the development of a high-quality software product. The evaluation of individual member's software design can be found in this Section. A more comprehensive discussion on the metrics of each individual software design can be found in Section 1.2. After evaluation based on the ISO/IEC 25010 quality model, we decided on choosing Lucas' design for implementation. This is because it scores the highest during software evaluation with ISO/IEC 25010 quality model.

Lucas implemented the Facade Design Pattern for the game architecture. We agreed as a team that this strategic choice streamlined the interface to complex game logic by encapsulating subsystems, thus providing a clean interface to client code. The rationale behind this decision was clear. It not only simplified current implementation challenges but also laid a robust foundation for future enhancements and scalability, aligning well with the project's long-term objectives. Furthermore, the approach to split the game logic into several handler classes exhibited a commitment to modular design principles. By breaking down the complexity of the game logic into manageable units, the system became inherently more maintainable, testable, and extendable. This decision highlighted Lucas' understanding of object-oriented design principles and its practical implications for software development. It sets it apart from the other implementations from other team members.. In the realm of object instantiation, Lucas opted for a CreatureName enumeration over a Creature class, demonstrating a sensible use of design patterns to simplify the implementation. By recognizing that multiple subclasses for different creatures were unnecessary due to minimal variations in attributes, Lucas made a practical decision that optimised code structure without sacrificing functionality. His design implementation adopted Model-View-Controller (MVC) architecture which showcased a commitment to architectural best practices. This design is consistent in the other software implementations by other team members. This implementation is strongly recommended and supported by all team members as the MVC design encourages a separation between the software's business logic and display which is the industry standard to simplify design decisions. This architectural choice not only promoted modularity by separating concerns but also leveraged JavaFX features for building responsive interfaces, aligning seamlessly with the project's technical requirements and future scalability goals. In the graphical implementation aspect, Lucas leveraged JavaFX Shape classes like Rectangle and Circle which can be observed across all team members' implementations, demonstrating a creative approach to leveraging existing libraries for

efficient development. By utilising built-in methods for rendering shapes and managing their positions, the graphical implementation was not only streamlined but also ensured compatibility and performance optimization. The decision to apply the Factory Method pattern for card creation highlighted the attention to design patterns for flexibility and maintainability. By encapsulating object creation logic, the system became more adaptable to handle different card types based on runtime data or configurations, ensuring a robust and scalable solution.

After discussion with all team members, we came to a unanimous decision that Lucas' design emerged as the most suitable choice for implementation based on its comprehensive alignment with the ISO/IEC 25010 quality model criteria. The assessment based on the ISO/IEC 25010 quality model criteria can be found in this Section as well as in Section 1.2. From strategic architectural decisions to meticulous attention to code quality and user interface aesthetics, Lucas demonstrated a holistic understanding of software development principles and a clear vision for delivering a high-quality software product.

SOFTWARE PRODUCT QUALITY								
FUNCTIONAL SUITABILITY	PERFORMANCE EFFICIENCY	COMPATIBILITY	INTERACTION CAPABILITY	RELIABILITY	SECURITY	MAINTAINABILITY	FLEXIBILITY	SAFETY
FUNCTIONAL COMPLETENESS FUNCTIONAL CORRECTNESS FUNCTIONAL APPROPRIATENESS iso25000.com	TIME BEHAVIOUR RESOURCE UTILIZATION CAPACITY	CO-EXISTENCE INTEROPERABILITY	APPROPRIATENESS RECOGNIZABILITY LEARNABILITY OPERABILITY USER ERROR PROTECTION USER ENGAGEMENT INCLUSIVITY USER ASSISTANCE SELF-DESCRIPTIVENESS	FAULTLESSNESS AVAILABILITY FAULT TOLERANCE RECOVERABILITY	CONFIDENTIALITY INTEGRITY NON-REPUDIATION ACCOUNTABILITY AUTHENTICITY RESISTANCE	MODULARITY REUSABILITY ANALYSABILITY MODIFIABILITY TESTABILITY	ADAPTABILITY SCALABILITY INSTALLABILITY REPLACEABILITY	OPERATIONAL CONSTRAINT RISK IDENTIFICATION FAIL SAFE HAZARD WARNING SAFE INTEGRATION

Figure 1.1: ISO/IEC 25010 (adapted from https://iso25000.com/index.php/en/iso-25000-standards/iso-25010)

1.1.1 Functional Completeness & Correctness (Completeness of Solution Direction)

Scale: 1 (lowest) to 5 (highest)

Assessing the completeness of the solution direction is paramount to ensure that all essential functionalities are encompassed within the design. This evaluation, aligned with the ISO/IEC 25010 quality model's criteria of functional completeness and functional correctness, seeks to determine the extent to which the proposed solution addresses the project's requirements without errors. By examining the comprehensiveness of the design in covering key functionalities, we can gauge the project's readiness for implementation and its potential to meet user expectations effectively.

Lucas (4) - This is because most of the game features were implemented in Sprint 2, however essential features to make the game functionally complete were not yet implemented. For example, the game’s logic to move the cards based on the flipped card was not implemented.

Brian (3) - This is because the cards implemented in Sprint 2 were able to be flipped back to demonstrate the flipping functionality which is acceptable for Sprint 2 but shouldn't be the case in the actual player's turn.

Sineth (1) - This is because, the functionality of winning the game was not completed in sprint 2, to which the set of functions do not cover the specified task and intended user's objectives.

Hafiz (3) - This is because the functionality of changing player's turn was completed but mainly for debug mode as it was implemented using a "next player" button to go to the next turn. It will only go to the next run when the user clicks on the "next player" button. Therefore, correctness might not be on par.

1.1.2 Functional Appropriateness (Rationale Behind the Chosen Solution Direction)

Scale: 1 (lowest) to 5 (highest)

Understanding the rationale behind the chosen solution direction is crucial for ensuring that the proposed approach aligns effectively with the project's objectives and requirements. This evaluation focuses on functional appropriateness, seeking to ascertain whether the chosen solution direction is well-suited to addressing the project's challenges and goals. By examining the reasoning behind each team member's approach, we can identify strengths, weaknesses, and potential areas for improvement, ultimately guiding decision-making towards the most suitable solution strategy.

Lucas (3) - The chosen solution direction of Lucas's design involves strong coupling between classes. For example, the MainAppController has very strong coupling between many different classes. Tight coupling can lead to a system where changes in one module necessitate changes in many others, which complicates maintenance and scalability.

Brian (4) - This is because the chosen functionalities and their implementations align well with the overall game concept and design. Each class has its own purpose and data holder classes are avoided.The purpose of each class in the design is easily understood by team members.

Sineth (1) - This is because the winning game functionality was not completed, as the ending of the game when the player returns to its cave did not complete the game by showcasing a different page indicating which player won.

Hafiz (2) - This is because it was implemented in a singleton design especially in the Main class.Using a singleton pattern can limit flexibility and testability, making the function less adaptable to changes or different contexts,

1.1.3 Appropriateness Recognizability (Understandability of Solution Direction)

Scale: 1 (lowest) to 5 (highest)

The understandability of the solution direction is vital for promoting effective collaboration and comprehension among team members. This evaluation examines the clarity and

coherence of the proposed solution, aiming to ensure that it can be easily grasped and communicated within the development team. By assessing the appropriateness of recognizability, we can identify any potential ambiguities or complexities in the solution direction and address them proactively to enhance overall project understanding and cohesion.

Lucas (2) - Lucas's solution direction involves using the JavaFX classes and the Facade Design Pattern which the group members are not familiar with. Therefore, the group will have to conduct tech spikes and group discussions to understand Lucas's proposed solution.

Brian (3) - This is because the code is not easily understood by team members initially but contains familiar classes that each member could identify the relationship with. There appears to be limited explanation required to other team members for them to understand the rationale behind most of the classes that were implemented.

Sineth (2.5) - This is because code is not complete, hence, other team members could not envision or understand the design of the final game. However, due to the code being unfinished, it was still understandable of what was written for submission due to its simplicity.

Hafiz (3) - This is because it is very understandable as it is just a singleton design but in terms of tracking specific functions and integrating may be difficult for others.

1.1.4 Modifiability (Extensibility of Solution Direction)

Scale: 1 (lowest) to 5 (highest)

Anticipating future extensions and enhancements is essential for ensuring the longevity and adaptability of the proposed solution. This evaluation focuses on the modifiability of the solution direction, assessing its ability to accommodate future changes and additions without requiring significant rework. By examining the extensibility of each solution approach, we can identify design patterns, architectural choices, or implementation strategies that facilitate seamless integration of new features or functionalities in subsequent project phases.

Lucas (3) - This is because the design is currently using JavaFX elements placed onto the scene. The more modifications in the design, the more complex it will become.

Brian (3.5) - This is because the codebase currently uses MVC architecture and is relatively easy to scale as it is implemented in combination with other design patterns such as Chain of Responsibility.

Sineth (1) - This is because the code has no structure due to its incompleteness. This is because of the individuals lack of knowledge on java and game design.

Hafiz (1) - This is because, using a singleton pattern limits modifiability because it restricts the system to a single instance of a class, making it hard to change or extend functionality. It introduces tight coupling and global state, complicating testing and maintenance

1.1.5 Maintainability (Quality of Written Source Code)

Scale: 1 (lowest) to 5 (highest)

The quality of written source code significantly impacts the maintainability and scalability of the software solution. This evaluation assesses coding standards, reliance on robust programming constructs, and avoidance of anti-patterns such as down-casting. By evaluating the maintainability of the source code, we can ensure that it is well-structured, comprehensible, and easily modifiable, facilitating efficient collaboration and future enhancements. The cyclomatic complexity is also measured using a plugin called “Code Metrics”. Cyclomatic complexity is a software metric used to gauge the complexity of a program's control flow by counting the number of decision points, such as loops, conditionals, and branches. It quantifies the number of linearly independent paths through the code, providing insights into potential difficulties in understanding, testing, and maintaining the software. By keeping cyclomatic complexity low, we are able to improve code quality, making it more manageable and less error-prone.

The below text is the categorization of cyclomatic complexity:

- **1 - 10**: Simple procedure, little risk
- **11 - 20**: More complex, moderate risk
- **21 - 50**: Complex, high risk
- **> 50**: Untestable code, very high risk

Lucas (4) - In the code base there are multiple responsibilities intertwined in single components, making bug fixes, updates, or enhancements could be more time-consuming and error-prone. However, the classes have a lowest overall cyclomatic complexity of 3.75 per class compared to the other prototypes.

Class	**Cyclomatic Complexity**
GUI	3
Player	3
Game	6
CreatureName	3

Brian (3.5) - The codebase employs “Don’t Repeat Yourself” (DRY) principles throughout the classes. Therefore, the code is relatively easy to scale and maintain as the complexity of other classes is hidden from each other. This prototype has an overall cyclomatic complexity of 5.57 per class.

Class	**Cyclomatic Complexity**
Controller	12
GUI	3
Display	4

Game	3
Board	5
DragonCard	3
Creature	9

Sineth (1.5) - This is because, even with the code's incompleteness, it still maintains the Single Responsibility Principle, by allocating each class a single responsibility. Following the “Don’t Repeat Yourself” (DRY) principles throughout the classes on top of it as well. This prototype has an overall cyclomatic complexity of 5.25 per class.

Class	**Cyclomatic Complexity**
Board	12
Creature	3
DragonToken	3
Player	3

Hafiz (2) - The cyclomatic complexity values for the classes creature (9), GUI (3), Hellocontroller (5), and player (3) indicate that they fall within the "Simple procedure, little risk" category. This suggests the source code is well-structured, easily maintainable, and presents minimal risk. Such low complexity improves code quality, making it more manageable, comprehensible, and less error-prone, facilitating efficient collaboration and future enhancements.
This prototype has an overall cyclomatic complexity of 5 per class.

Class	**Cyclomatic Complexity**
Creature	9
GUI	3
HelloController	5
Player	3

1.1.6 User Engagement (Aesthetic of the User Interface)

Scale: 1 (lowest) to 5 (highest)

The aesthetic of the user interface plays a crucial role in enhancing user engagement and satisfaction. This evaluation focuses on aspects such as visual appeal, usability, and intuitiveness, aiming to create a positive user experience. By assessing the user interface aesthetics, we can identify design choices, colour schemes, layout patterns, and interactive elements that contribute to a compelling and user-friendly interface, ultimately improving the overall usability and attractiveness of the software product.

Lucas (5) - The game interface design effectively combines visual clarity, user-friendly layout, and thematic consistency to enhance player engagement and ensure smooth gameplay. Colour-coded tracks and creature cards help in quick identification and decision-making, while the clear display of the current player and interactive buttons like "Next Player" facilitate easy navigation and interaction. The circular board and minimalist aesthetic keep the focus on gameplay without overwhelming players with details, making the game accessible and enjoyable for a broad audience. These elements are thoughtfully integrated to support an intuitive and immersive gaming experience.

Brian (2) - The game currently is not engaging to players as the features are limited to flipping and randomising the cards due to the chosen Sprint 2 features being flipping and initial set-up of the board.

Sineth (3.5) - The User Interface is aesthetic and engaging, as it includes everything required as well as promoting the colours of the game. The two shades of circles covering the chit cards and volcano cards are very attractive. Moreover, the colours interact with each other smoothly, providing a compelling and user-friendly interface.

Hafiz (3) - The game UI is not as aesthetically pleasing but easily understandable with the player turn display and button for moving to next turn can be easily seen. The volcano cards are also labelled with creatures though there are no colour distinction between different creature volcano cards

1.2 Review of Sprint 2 Tech-based Software Prototypes

In this section, we will conduct a comprehensive review of each Sprint 2 tech-based software prototype, utilising the comparison criteria outlined earlier. Each prototype will undergo evaluation based on criteria such as completeness of the solution direction, rationale behind the chosen solution, understandability, extensibility, quality of written source code, and aesthetic of the user interface. The review process will be conducted as a group, with the author of each prototype providing clarifications as needed, but not explanations. By critically assessing each prototype against these criteria, we aim to identify strengths, weaknesses, and key findings to inform further development and decision-making.

1.2.1 Lucas' Prototype

Lucas' prototype included the following classes in the UML class diagram.

1. JavaFX Color, Rectangle, Circle, Arc, Text, Button, Stage, Scene, Application
2. MainApp
3. MainAppController
4. Player
5. Game
6. CardFactory
7. DragonCard
8. PlayerHandler
9. DragonCardHandler
10. VolcanoCardHandler
11. VictoryHandler

12. MovementHandler
13. CreatureName
14. VolcanoCard
15. Cards

Functional Suitability (1 is lowest, 5 is highest):
Functional Completeness (4) - The implementation of the game has most of the key game features like movement of the dragon token, volcano card and dragon card randomisation and player turns. However the essential game logic is not yet implemented.

Functional Correctness (3) - The implementation is mostly correct however it was discovered that there were several bugs such as the dragon token not being able to move into its own cave and that the volcano card randomisation is not correct.

Functional Appropriateness (5) - The selected functionalities and their implementations are well-suited to the game's requirements and objectives, aligning effectively with the overall game concept and design.

Performance Efficiency (1 is lowest, 5 is highest):
Time Behaviour (3) - The building of the project into an executable takes about 10 seconds, which is slow. However, when the executable is built, running it only takes less than 0.5 seconds.

Resource Utilisation (5) - The executable using Java efficiently utilised system resources such as memory and CPU to ensure smooth gameplay without excessive resource consumption.

Capacity (2) - The executable has a large file size, this might be because of the numerous libraries that are packaged together with the executable like JavaFX and some images.

Compatibility (1 is lowest, 5 is highest):
Co-existence (5) - The executable can perform its required functions efficiently while sharing a common environment and resources with other software, without negatively impacting any other software.

Interoperability (NA) - The executable exists on its own and doesn’t depend on any other software for data.

Interaction Capability (1 is lowest, 5 is highest):
Appropriateness recognizability (4) - The UI of the game has a simple design with simple colours. The dragon tokens are also distinguishable from each other as they have a unique Pac Man shape with 4 different colours.

Learnability (1) - The executable/game does not have a tutorial or guide on how to play the game. First time players will not know what to do.

Operability (4) - The game has several buttons on the menu for the player to control the player’s movement or change the state of the game.

User error protection (5) - The game's source code contains guardian code to prevent user errors and protect against potential mistakes by players.

User engagement (2) - The game does not have interesting visuals or sound effects, so this might not be very interesting to players.

Inclusivity (1) - The game is not targeted towards people with diverse backgrounds, abilities, and needs. For example, colour blind people might find it difficult to play the game due to the usage of various colours.

User assistance (1) - The game does not provide any type of help or support features.

Self-descriptiveness (1) - The game does not present any necessary information for players to make its capabilities and usage clear without excessive interaction with help resources.

1.2.2 Brian's Prototype

Brian's prototype included the following classes in the UML class diagram.

1. JavaFX Scene
2. JavaFX Event Handler
3. JavaFX Image
4. JavaFX Application
5. JavaFX Button
6. JavaFX Text
7. JavaFX Rectangle
8. JavaFX Initializable
9. Main
10. GUI
11. Display
12. Controller
13. Game
14. Board
15. VolcanoCard
16. Player
17. DragonCard
18. Creature

Brian's implementation of a Controller Class offers several advantages, including modular design, easier maintenance, better testability, and promotion of reusability. However, it introduces increased complexity and potential tight coupling. Despite these drawbacks, it was implemented to enable easier extensibility and scalability, crucial for upcoming sprints. His Game Class implementation brings advantages such as encapsulation, modifiability, facilitation of communication, and support for a one-to-many relationship. However, it also increases complexity and the risk of tight coupling if not managed properly. Despite these challenges, it was implemented to simplify communication and enhance modifiability of the game logic. Choosing composition over aggregation for the relationship between Board and VolcanoCard offers flexibility, simplicity, and code reusability. This decision aligns with the

game's requirements for clearer ownership and flexibility. Implementing the Board Class as a Singleton ensures a single instance of the board per game, simplifying management and preventing issues with multiple instances. This choice was made to simplify board management and ensure consistency across the game. Brian has avoided using inheritance for the dragon cards but instead with an enumeration class to simplify the design and avoid the creation of multiple data holder classes. This approach leads to a simpler design without sacrificing functionality. Establishing a one-to-many relationship between Game and Player simplifies player management and avoids redundancy in player classes. This decision streamlines game logic and ensures efficient player management. Defining specific cardinality for DragonCard and VolcanoCard ensures consistency and manageability in the game. This approach facilitates future extensions and ensures the correct number of cards in the game. Implementing design patterns such as Chain of Responsibility and Facade enhances code maintainability and flexibility. These patterns were chosen to adhere to design principles and improve overall code quality. In conclusion, the decision to implement the discussed design choices appears justified, as they align with the requirements of the game project, facilitate future extensions, and promote maintainability and scalability. Each aspect contributes to a well-structured and robust software architecture for the "Fiery Dragons" game.

Functional Suitability (1 is lowest, 5 is highest):
Functional completeness (2) - The implementation covers 2 key functionalities required for the "Fiery Dragons" game, including setting up the initial game board and flipping dragon cards. The other functionalities (moving dragon tokens, changing turns, and determining the winner) are not implemented.

Functional correctness (3.5) - The implementation seems to correctly execute the defined functionalities of flipping as described in the game documentation. However, it is discovered that the cards implemented in Sprint 2 were able to be flipped back to demonstrate the flipping functionality. It is deemed acceptable for Sprint 2 but this would need modification for Sprint 3. Thorough testing would be necessary to confirm the correctness of each function's behaviour.

Functional appropriateness (4) - The chosen functionalities and their implementations are appropriate for the game's requirements and objectives, aligning well with the overall game concept and design.

Performance Efficiency (1 is lowest, 5 is highest):
Time behaviour (4) - The performance of the system in terms of response time and processing speed is tested to be within 0 to 0.3 s. This depends on factors such as the complexity of the game logic and the efficiency of the code implementation.

Resource utilisation (4) - The system utilised system resources such as memory and CPU efficiently to ensure smooth gameplay without excessive resource consumption. The laptop is relatively cool and smooth during the execution of the game.

Capacity (4) - The system is able to handle a reasonable number of flipping actions within short intervals (due to the flipping function being chosen) and game elements without significant degradation in performance or functionality.

Compatibility (1 is lowest, 5 is highest):
Co-existence (5) - The software product would be able to coexist with other software by performing its required functions efficiently while sharing a common environment and resources with other software. This is because the software is coded into a java module which can be used by other softwares to extend the functionality without much refactoring or detrimental impact on the current software. Therefore, there would be seamless integration within the broader software ecosystem.

Interoperability (2.5) - The system is able to interact and communicate effectively with external systems or components, such as mouse interactions with user interfaces. However, it does not support keyboard interactions to support seamless operation.

Interaction Capability (1 is lowest, 5 is highest):
Appropriateness recognizability (4) - The user interface and interaction design is intuitive and recognizable, allowing players to easily understand and navigate the game, especially the dragon cards as they appear highlighted when hovered above.

Learnability (3) - The system is relatively easy for new users to learn, with clear instructions on which buttons to press. However, there are no tutorials provided to help them understand the game mechanics.

Operability (4) - The system is easy to operate, with straightforward controls in the form of mouse controls and commands that facilitate smooth gameplay.

User error protection (3) - The system incorporates features or mechanisms to prevent or mitigate user errors such as in the form of guardian code. However, further features such as confirming critical actions or providing error messages for incorrect inputs can be implemented.

User engagement (2) - The game is not engaging to players as the features are limited to flipping and randomising the cards. Features such as animations, sound effects, and interactive elements should be added to enhance the gaming experience.

Inclusivity (1) - The system is currently not accessible to a diverse range of users as it does not incorporate the feature of hearing loop or provide support to those with disabilities. Incorporating features such as assistive listening system, adjustable difficulty levels or alternative control options would improve the inclusivity.

User assistance (2) - The system indicates that the dragon cards are clickable. However, it should provide more assistance or guidance to users in the form of tutorials, tooltips, hints, or help menus.

Self-descriptiveness (4) - The system is pretty self-explanatory and provides clear visualisation and instructions to users about its current state or actions, reducing the need for external documentation or assistance.

1.2.3 Sineth's Prototype

Sineth's prototype included the following classes in the UML class diagram.

1. Game
2. Board
3. Dragon Card
4. Creature
5. Controller
6. Display
7. Dragon Token
8. Cave
9. Position
10. Volcano Card

Sineth's software design involves classes such as dragon token class and board classes. He uses inheritance and enumeration in the game design. The rationale behind his design is as follows.

Dragon Token Class

In the 'Fiery Dragons' game, players need unique tokens to represent their positions on the board, which also helps track their scores. The choice between a single Player class or separate classes for Player and Dragon Token was considered. A single Player class might simplify the code, but it would violate the Single Responsibility Principle by mixing responsibilities. Instead, having distinct classes for Player and Dragon Token allows the Player class to handle player data, and the Dragon Token class manages the representation of tokens on the board, enhancing modularity and flexibility. This separation allows for easier modification and extension. The relationship between the Player and Dragon Token classes is aggregation because each class can function independently but work together for gameplay.

Board Class

The Board class manages the game's physical aspects, like flipping 'chit' cards and placing different card types on the board. Though the Board class is indirectly connected to the Dragon Token class via the Position and Game classes, combining the Board and Game classes was avoided to respect the Single Responsibility Principle. Keeping them separate simplifies maintenance and modifications, especially for extensions. The relationship between the Board and Game classes is aggregation due to their independent existence. The cardinality is one-to-one since multiple games can be played using the same board, but each game uses one board at a time.

Inheritance and Enumeration

Initially, the game's creatures were designed using inheritance, where the Creature class served as the base for various subclasses representing different creatures. However, since these creatures do not have unique abilities, an enumeration was more appropriate. An enumeration class better fits the game design by offering constants to represent creatures and aiding in matching them to volcano cards, improving code readability and maintainability.

Functional Suitability (1 is lowest, 5 is highest)

Functional completeness (1) - The game needs to ensure that all features, such as the set up of the initial game board, flipping of dragon ‘chit’ cards, movement of dragon tokens based on their current position as well as the last flipped dragon 1 card, change of turn to the next player and winning the game, are fully implemented and align with the requirements. The only functionality implemented was the set up of the initial board game, hence the score.

Functional correctness (1) - Winning the game functionality was not implemented correctly and could not be executed. It's essential that each feature, like moving Dragon Tokens or calculating scores, works accurately according to the established game rules. The separation of responsibilities into different classes supports this by making each class responsible for a specific task.

Functional appropriateness (4) - The chosen functionalities and their implementations are appropriate for the game's requirements and objectives, aligning well with the overall game concept and design. The distinct separation of Player and Dragon Token classes ensures each class has a focused responsibility, allowing the features provided to be aligned with their intended use, thus making them more relevant and intuitive.

Performance Efficiency (1 is lowest, 5 is highest):
Time behaviour (1) - Game does not execute. Hence, the response time of the game cannot be calculated when performing.

Resource utilisation (1) - Game does not execute. Hence, the amount and types of resources cannot be calculated when performing.

Capacity (1) - Game does not execute. Hence, not meeting the requirement.

Compatibility (1 is lowest, 5 is highest):
Co-existence (2) - The game's modular design ensures it can coexist with other applications, as its efficient handling of game data and graphical resources avoids unnecessary conflicts with other processes.

Interoperability (1) - The system can not exchange information with other external systems or components, such as mouse interactions with user interfaces. As the handlers were not able to be implemented.

Interaction Capability (1 is lowest, 5 is highest):
Appropriateness recognizability (2.5) - The game board is recognisable and easy to understand and navigate through. However, small functionalities such as the chit cards being highlighted as they are hovered over, along with the other buttons such as move forward and back, could have improved the recognizability of the game board.

Learnability (3) - The system is relatively easy for new users to learn, with clear instructions on which buttons to press. However, there are no tutorials provided to help them understand the game mechanics.

Operability (4) - The game's classes like Board and Player are designed to be operable, with straightforward interfaces for managing gameplay, ensuring easy control for the user.

User error protection (1) - The system does not involve the prevention of users against operation errors. These features need to be implemented.

User engagement (1.5) - Apart from the game board's aesthetic, there is no additional functions and information in an inviting and motivating manner encouraging continued interactions. Additional functionalities like narration, sound effects and animation would definitely increase user engagement, and need to be implemented.

User assistance (1) - There is no guidance or tools to help the users with the functionality and use of the game. The buttons are not even clickable.

Self-descriptiveness (3) - From observing the game board, it is self-explanatory on how to play the game. The only confusion would be the lack of instructions on flipping the ‘chit’ cards. Other than that, the game buttons of moving forward and back showcase the simplicity of the game, not needing excessive amounts of external information.

1.2.4 Hafiz’s Prototype

Class Diagram:

1. JavaFX Button
2. JavaFX Text
3. JavaFX Rectangle
4. JavaFX Scene
5. JavaFX Circle
6. JavaFX Color
7. JavaFX Stage
8. JavaFX
9. HelloController
10. MainApp
11. GUI
12. Player
13. Game
14. Creature
15. VolcanoCard
16. ChitCard

In designing the architecture of the game application, several design principles and patterns have been employed.

VolcanoCard Class Addition:
Initially, the absence of a VolcanoCard class seemed appropriate, as the cards appeared fixed. However, to adhere to the Open Closed principle and ensure flexibility for future iterations, the VolcanoCard class was introduced. By defining attributes like creature type, location, occupancy status, and cave association, the class lays the groundwork for potential features like card shuffling and enhanced cave functionality. This anticipatory design allows for seamless expansion while maintaining simplicity.

Singleton Pattern for Game Class:
The Game class was implemented as a Singleton to provide global accessibility to its instance across the application. This design choice facilitates efficient communication and coordination between various components, enhancing overall system cohesion. Benefits such as optimal resource utilisation and lazy initialization contribute to improved performance and resource management. However, careful consideration is necessary to address potential challenges like complex data flow and concurrency issues.

Composite Pattern for UI Organization:
The composition design pattern was employed to structure the UI components within the helloController class. By encapsulating JavaFX components and promoting modularity, code reuse, and separation of concerns, this approach fosters maintainability and flexibility. It enables easy customization and modification of UI elements without compromising the overall architecture. Despite potential complexities in managing numerous components, the benefits of modularity and code clarity outweigh the challenges.

Separation of Concerns:
By delineating responsibilities between JavaFX for GUI rendering and the controller class for application logic, a clear separation of concerns is achieved. This design choice enhances testability, manageability, and collaboration among developers. With the controller class focusing solely on user input, logic execution, and UI component coordination, the codebase becomes more organised and comprehensible. This division fosters cleaner code, easier maintenance, and improved teamwork.

Functional Suitability (1 is lowest, 5 is highest):
Functional completeness (4) - The inclusion of classes like Game, Creature, VolcanoCard, and ChitCard seems to cover the essential aspects required for the game's functionality and the individual requirement which is change of player turn. The game is also executable.

Functional correctness (2) - The Game may fulfil the requirements for Sprint 2 but not implemented correctly as it follows singleton design. The absence of certain classes or features like the ChitCards, playerToken, VolcanoCards class and ChangePlayerTurn class may affect the correctness. These features were hardcoded in the scenebuilder.

Functional appropriateness (2)- The functional appropriateness score of 2 out of 5 reflects issues such as incomplete features, potential performance challenges, and UI complexities. While solid design principles and patterns are employed, the current implementation falls short in execution.

Performance Efficiency (1 is lowest, 5 is highest):
Time behaviour (3) - The time behaviour of the game application is moderately efficient. The introduction of the Singleton pattern for the Game class ensures efficient communication and coordination across components, which helps in reducing the overhead of instance creation and initialization. However, potential challenges with complex data flows and concurrency issues might lead to delays or inefficiencies in certain operations, preventing the application from achieving optimal performance. Improvement in handling these issues could enhance the overall responsiveness and speed.

Resource utilisation (3) - Resource utilisation in the game application is quite efficient. The Singleton pattern ensures optimal resource utilisation by maintaining a single instance of the Game class, which conserves memory and processing power. Additionally, the Composite pattern for UI organisation promotes modularity and code reuse, which can lead to better management of system resources. However, while these design patterns are effective, there may still be some room for optimization, particularly in managing numerous UI components and preventing resource leaks.

Capacity (2) - The game can manage more load and features, but not without limits. Using the Open Closed concept, the VolcanoCard class allows for card shuffling and cave feature upgrades. This anticipatory design allows application expansion without major reworking. Due to performance difficulties and resource limits, the existing implementation may struggle under heavy demand or with major feature enhancements.

Compatibility (1 is lowest, 5 is highest):
Co-existence (2) -Coexistence is moderate in the architecture. Every design decision, like adding the VolcanoCard class and using the Singleton approach for the Game class, integrates in well. This lets new features be introduced without interrupting system operation. Minor tweaks may be needed to suit future versions.

Interoperability (2) - The architecture scores moderately for interoperability. The Composite method for UI structure and explicit separation of responsibilities enhance modularity and cooperation among components, although integration with other systems or future changes may be difficult. Communicating and working with external dependencies requires careful planning.

Interaction Capability (1 is lowest, 5 is highest):
Appropriateness recognizability (4) - The user interface and interaction design of the game is characterised by its intuitive and recognized nature, enabling players to effortlessly comprehend and move through the game. Notably, the dragon cards are illuminated when the cursor is placed over them, further enhancing their visibility and facilitating interaction. Also the display of current players helps in the recognisability.

Learnability (3) - The design is relatively easy to learn as the UI only has a shuffle chit cards button and also changes to the next player button though no instruction or documentation provided.

Operability (4) - The system is easy to operate with the buttons provided

User error protection (1) - The design does not provide user error protection.

User engagement (2) - The game was not implemented with the full functionality, for now it is only able to change players and randomise chit cards.

Inclusivity (1) - The design lacks explicit specifications for inclusion elements, nonetheless, it is essential to include inclusivity concerns into the implementation process, particularly by including accessibility features to cater to people with impairments.

User assistance (1) -User assistance features like tooltips or in-game instructions are not mentioned but can be implemented to aid users in understanding the game mechanics.

Self-descriptiveness (2) - The system right now is pretty self explanatory with users being able to only change player turn and randomise the chitcards. Therefore, no description is needed

1.3 Sprint 3 Tech-Based Prototype Planning

To proceed effectively with the development of the tech-based prototype for Sprint 3, we carefully consider the strengths and weaknesses of each Sprint 2 prototype. We assessed the ideas and elements from each prototype to determine which ones to carry forward and identify new ideas or elements required for improvement. By strategically selecting components and enhancements, we aim to create a robust and comprehensive prototype that covers the entire game functionality while addressing any deficiencies observed in Sprint 2. After a detailed discussion within the team, the team has decided to continue the extension of the game using the base code from Lucas' design as we deemed that it best reflects the appropriate use of design patterns and object-oriented industry practices.

2. Object-Oriented Design

In this section, we delve into the object-oriented design aspect of the Fiery Dragon's game project. We begin by completing Class-Responsibility-Collaboration (CRC) cards for six of the main classes in our consolidated design. Each CRC card provides a brief description of the purpose of the chosen class, laying the foundation for a clear understanding of the class responsibilities and collaborations within our system. Additionally, we present a Class Diagram encompassing class names, attributes, methods, relationships between classes, and cardinalities of all required classes in our design. Through this structured approach to object-oriented design, we aim to establish a solid framework for the implementation phase while fostering effective collaboration and communication among team members.

2.1. Class-Responsibility-Collaboration card

In the process of designing the Fiery Dragon's game, six key classes have been identified, each contributing distinct responsibilities to the overall system. The completion of Class-Responsibility-Collaboration (CRC) cards for these classes is essential for defining their purposes, responsibilities, and interactions within the game. By succinctly describing the role and function of each chosen class, we establish a clear understanding of their contributions to the game's functionality and facilitate effective collaboration among team members during the design and implementation phases.

Game Class

The Game class serves as the central orchestrator within the game environment, managing various aspects of gameplay. Its primary responsibilities include placing players' tokens onto their designated caves, orchestrating player turns, handling player movement across the game board, and processing button clicks by forwarding them to the appropriate handlers. By coordinating interactions between players, tokens, caves, and user interface elements

such as JavaFX Buttons, the Game class ensures smooth gameplay progression and facilitates the core mechanics of the game. It acts as the backbone of the game logic

Game	
Place the players' tokens onto their respective caves	DragonToken & Cave
Manage player turns	Player
Handle player movement	DragonToken & Player
Process button clicks and forward them to the appropriate handlers	JavaFX Button

MainAppController Class

The "MainAppController" class acts as the central hub for controlling the initialization and management of the game within a JavaFX application. Its primary responsibilities include initialising the game environment, capturing player changes, and handling the game victory conditions as well as transitions of pages to winner page. By collaborating with other classes such as JavaFX Rectangle, Player, and JavaFX Scene, the MainAppController orchestrates the setup and presentation of game elements, ensuring a smooth and engaging user experience. Essentially, the MainAppController class serves as the entry point and coordinator for various game-related tasks, streamlining the initialization process and facilitating user interaction with the game interface.

MainAppController	
Initialise and set up UI components	JavaFX Rectangle, Circle, Image, Buttons
Listen for Player Changes	Player
Handle Game Victory Conditions & Transition to winner page	Player & JavaFX Scene

CardFactory Class

The "CardFactory" class plays a vital role in managing the creation, randomization, and distribution of cards within a game system. Its main responsibilities include initialising the cards (ensuring their random distribution for gameplay variability), managing card inventory, and providing a mechanism for retrieving information about specific cards when needed. By collaborating with instances of the "DragonCards" class, the CardFactory efficiently generates and administers the diverse array of cards required for gameplay. Whether it's setting up the initial deck, shuffling cards for each round, or facilitating card draws during gameplay, the CardFactory class maintains the integrity and dynamic nature of the game's card-based mechanics. Overall, it serves as the backbone for managing the lifecycle of cards within the game, ensuring a fair and engaging experience for players.

CardFactory	
Create Cards	DragonCards
Manage Card Inventory	DragonCards
Provide Card Information	DragonCards

DragonCardHandler Class

The "DragonCardHandler" class is instrumental in managing the behaviour and interactions associated with dragon cards within the game environment. Its primary responsibilities revolve around assigning creatures to these dragon cards, facilitating the flipping mechanism and storing the button-to-card mapping, which reveals the creature names and numbers. By collaborating with a Creature class, which includes Spider, Bat, Salamander, Baby Dragon, and Pirate Dragon, the DragonCardHandler ensures that each dragon card is populated with an appropriate creature. Additionally, the collaboration with JavaFX Button indicates the class's involvement in user interface interactions, allowing players to interact with dragon cards to flip them through button-based controls. Overall, the DragonCardHandler class acts as a crucial component in managing the dynamic interactions and outcomes associated with dragon cards.

DragonCardHandler	
Assign creatures to cards	CreatureName & DragonCards
Manage card flipping	JavaFX Button & DragonCards
Store Button-Card Mapping	JavaFX Button & DragonCards

VolcanoCardHandler Class

The "VolcanoCardHandler" class serves as a pivotal component in managing the behaviour and interactions related to volcano cards within the game system. Its primary responsibilities include initialising volcano cards, assigning the creatures to these cards as well as managing the volcano cards mapping to their positions. By collaborating with JavaFX Rectangle for graphical representation and a range of creatures such as Spider, Bat, Salamander, Baby Dragon, and Pirate Dragon, the VolcanoCardHandler ensures that each volcano card is populated with an appropriate creature.

VolcanoCardHandler	
Initialise volcano cards	JavaFX Rectangle
Assign creatures to Volcano Cards	CreatureName
Manage Volcano Cards Mappings	JavaFX Rectangle

MovementHandler Class

The "MovementHandler" class plays a crucial role in managing player movement and interactions within the game environment. Its primary responsibilities include processing button inputs (when a user clicks on the dragon cards to flip it), verifying if the dragon cards match the volcano cards on which a player is currently positioned, handling the initial and subsequent moves as well as managing scenarios where cards don't match. By collaborating with JavaFX Button for user interface interactions and the DragonCards class for accessing card-related information, the MovementHandler ensures seamless gameplay flow and enforces game rules regarding card matching. This class essentially facilitates player navigation through volcano cards, ensuring that their movements align with the game's mechanics and objectives. Additionally, it provides a mechanism for checking and resolving interactions between dragon and volcano cards.

MovementHandler	
Process button clicks	JavaFX Button
Check Matching Cards	DragonCards
Handle Initial and Subsequent Moves	DragonToken & DragonCards
Manage No-Match Scenarios	DragonToken & DragonCards

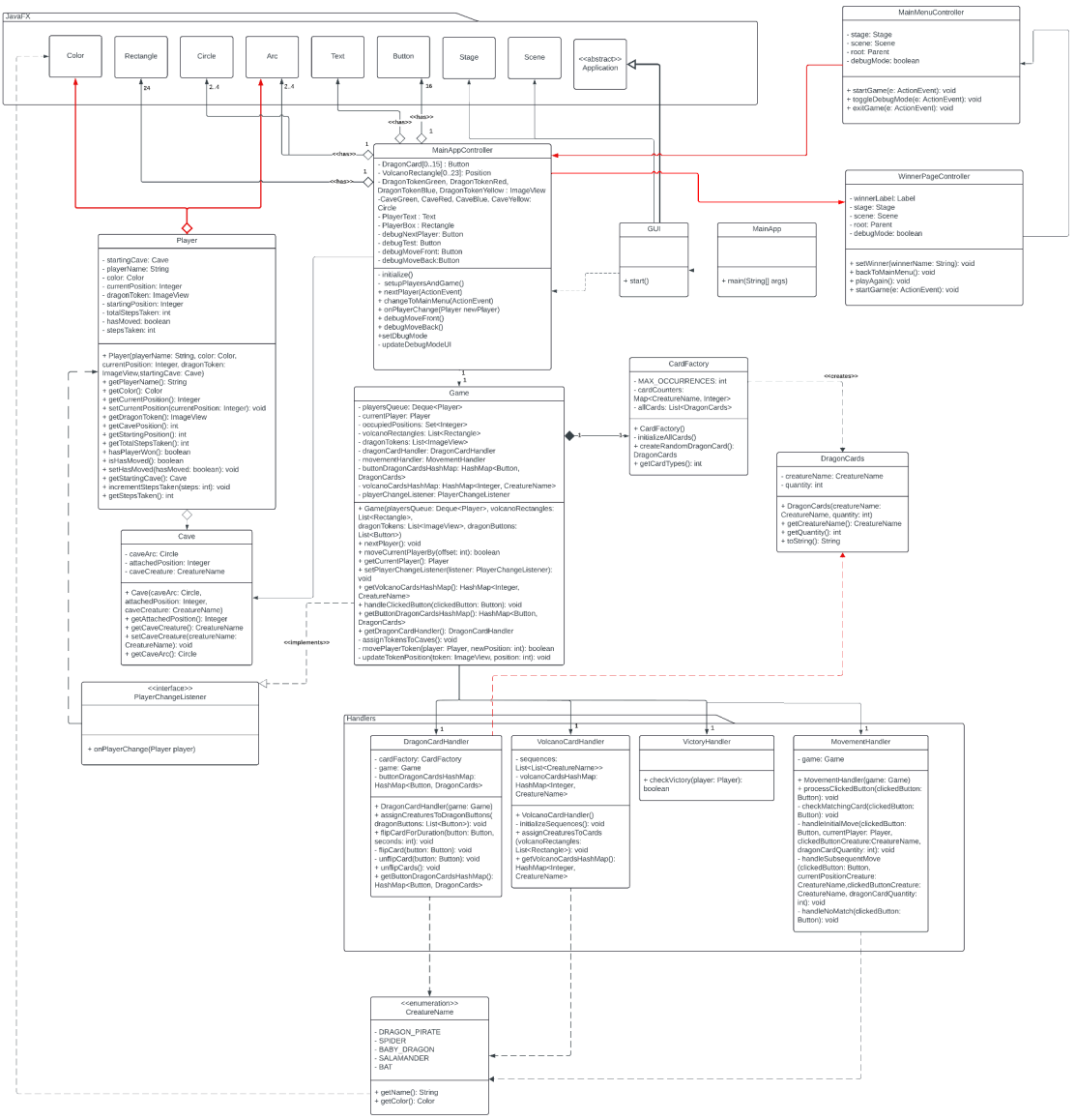
2.2. Class Diagram

In crafting the Class Diagram for the Fiery Dragon's game, we have meticulously outlined the essential components of our system, including class names, attributes, methods, relationships between classes, and cardinalities. This visual representation serves as a blueprint for the organisation and structure of our codebase, enabling a clear understanding of the interactions and dependencies among different classes. By carefully considering the design of our system before diving into coding, we can anticipate interaction patterns and ensure a smoother implementation process, ultimately leading to a more robust and efficient game development workflow.

Based on previous feedback and the ideas of our group, we have improved on the class diagram and have modified it to reflect the current state of the source code. For example, the abstract card class is deleted due to it being a data holder class and just adds unnecessary

complexity to the game. The diagram can also be accessed through the Lucidchart link below.

Lucidchart Link:
https://lucid.app/lucidchart/a9b8f420-f1f4-455d-8e87-27855af1f369/edit?viewport_loc=-1795%2C-4487%2C3282%2C1720%2Ch_Cy4_7XBG-K&invitationId=inv_1dbd4ba3-87b7-4078-92b0-ada97597c17b



3. Tech-based Software Prototype

3.1 Build Instructions

In this section, we provide comprehensive instructions for building the Fiery Dragon game executable on the Windows platform, however these instructions should work for any platform. These instructions serve as a guide for developers and stakeholders alike,

ensuring a smooth setup process and enabling easy access to the software prototype. To begin, it is recommended to have "Oracle OpenJDK version 22" module SDK as well as JavaFX version 21 installed on the Windows system to ensure successful execution of the program. You can download Liberica JDK as well as it includes JavaFX out of the box. Below are the two ways of building the project along with step-by-step instructions for building and running the executable:

We have included a script that makes building the project executable easy and simple using Maven. You can watch this YouTube tutorial to follow along the steps.

By following these steps, users can seamlessly build and run the Fiery Dragon game executable on the Windows platform, leveraging Oracle OpenJDK version 22 for optimal performance and compatibility.

You can watch the tutorial here, it includes two ways to build the project. One way uses Maven to build and the other one uses IntelliJ to build it.

2024-05-14 22-54-40.mkv

Maven Build Instructions:

1. Download Maven and ensure that the installation is successful with mvn -version.
2. Navigate to the Project Directory
3. Open the terminal and run the following commands:
    a. mvn clean compile
    b. mvn javafx:run
    c. mvn javafx:jlink