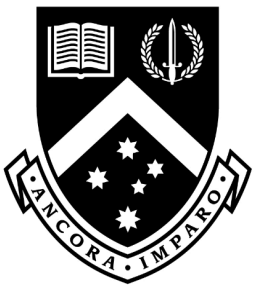


MONASH University

FIT3077 | Sprint 3 Group Assessment Report | S1 2024

Members: Gde Putu Guido, Timothy Suria, Andy Tay, Sayyidina Malcolm

Authored by: Team 098

Tutorial: Tuesdays 12-2 pm

This assignment was made to provide a learning experience and understanding from our eyes of the subject of FIT3077 and also to fulfill the requirements of the assignment (submission). We do understand that there might be mistakes in writing, and we apologize for that.

Table of Contents

Disclaimer

Our group will try to make this report so that it represents a comprehensive and objective assessment of each team member's Sprint 2 tech-based software prototype for the FIT3077 Software Engineering and Architecture Design unit. Our team is committed to providing constructive feedback and thorough evaluations based on the ISO/IEC 25010 quality model.

The assessments are intended to highlight strengths and identify areas for improvement to ensure the highest standards of software quality and functionality. Please note that the evaluations are based on the information provided in the respective design reports and our current understanding of best practices in software engineering.

Any recommendations for improvement are made with the goal of fostering learning and development within our team. Thank you for your attention to our report so far, and now, happy reading!

Assessment Criteria & Explanation

Assessment Criteria for Evaluating Tech-Based Software Prototypes

To evaluate the tech-based software prototypes developed by each team member, we established a comprehensive set of assessment criteria. We based our criterias ISO/IEC 20510 Quality Model and also the mentioned criterias in the Sprint 3 brief. These criteria ensure a thorough and objective analysis of each prototype's performance, functionality, and overall quality. **The key criteria are as follows:**

1. Completeness of the Solution Direction

- Functional Completeness: Evaluate the extent to which all key functionalities are covered in the design and implemented.
- Functional Correctness: Assess how accurately the implemented features work according to the specified requirements.

2. Rationale Behind the Chosen Solution Direction

- Functional Appropriateness: Analyze the rationale and justification for the chosen solution direction, ensuring it aligns with the project goals and requirements.

3. Understandability of the Solution Direction

- Appropriateness Recognizability: Assess how easily the solution direction can be understood by team members, stakeholders, and users. This includes the clarity of design, documentation, and communication.

4. Extensibility of the Solution Direction

- Modifiability: Evaluate the potential for extending the solution direction in anticipation of future enhancements or additional features planned for Sprint 4. This includes modularity, flexibility, and scalability of the design.

5. Quality of the Written Source Code

- Maintainability: Analyze the quality of the source code, including adherence to coding standards, use of design patterns, and avoidance of anti-patterns such as excessive case analysis or down-casts.

Additional Considerations (that our team try to adhere to too)

- Overall Project Understanding: Assess each team member's understanding of the project objectives, requirements, and constraints.
- Feedback Incorporation: Evaluate how well each prototype has incorporated feedback from previous reviews or testing phases.
- Future Improvements: Identify potential areas for improvement and scalability for future development phases.

A. Andy Tay's (31989934) Prototype

Assessment of Andy Tay's Sprint 2 Tech-based Software Prototype

1. Functional Suitability

- Functional Completeness:

- The prototype effectively demonstrates the key functionalities of initializing and moving tokens, setting up the board, and interacting with chit cards.
- The simplification to a 1 to 1 cardinality for the prototype is understandable for initial development. However, the full game functionality requires a 2 to 4 cardinality, indicating a partial gap in the current functional completeness.

- Functional Correctness:

- The described functions (initializeCellPositions, externalSetup, initializeCaves, initializeChit, initializeToken, moveToken) are logically sound and seem to provide accurate results based on the provided descriptions.
- Results are tested and executed by the team, this is done to confirm the correctness of these functionalities and to strengthen this assessment.

- Functional Appropriateness:

- The focus on token movement and chit-card functionalities is appropriate, as these are central to demonstrating the game's core mechanics.
- The choice to not use inheritance, despite repeating attributes, is justified by the current simplicity of the design and the desire to manually place objects.

2. Performance Efficiency

- Time Behavior:

- Andy's design report on sprint 2 and overall tech-based work does not detail performance metrics such as response times or throughput. If given the chance, future iterations would include an assessment of the initialization time for the board and the token movement to provide a better understanding of time behavior.

- Resource Utilization:

- No specific details are provided on resource utilization. Including this information would be beneficial, especially for understanding how the prototype performs under different loads, such as with multiple players.

- Capacity:

- Andy's current implementation is limited to single-player interactions. And since it is this way, testing the system's capacity to handle the full 2-4-player setup will be necessary for future iterations but not doable for our team now.

3. Compatibility

- Co-existence:

- Andy's prototype does not mention interaction with other systems or software, which is acceptable for its current standalone nature. However, considering potential integration points could be valuable for future development.

- Interoperability:

- Due to the scope, the design does not address interoperability with other systems. This could be an area for future improvement, especially if the game is to be expanded or integrated with other platforms.

4. Interaction Capability

- Appropriateness Recognizability:

- The functions and classes are clearly named, making their purposes recognizable for users and developers. Including more detailed documentation would further enhance this aspect.

- Learnability:

- The provided design rationale, tech-based prototype, and class descriptions aid in understanding the system. However, adding a user guide or tutorial would significantly improve learnability.

- Operability:

- The operations of the system are straightforward and well-explained, focusing on moving tokens based on chit cards. User feedback on the ease of use would be valuable for future iterations.

- User Error Protection:

- The design does not detail mechanisms for protecting against user errors. This is an area that could be enhanced to improve the overall user experience.

- User Engagement:

- Andy's report and tech prototype lack information and implementation on the aesthetics or engagement of the user interface. Future iterations should include user interface design considerations to make the game more inviting and motivating.

- User Assistance:

- The prototype could benefit from user assistance features such as tooltips or in-game help to guide users.

- Self-descriptiveness:

- While the system's functions are clear, further documentation and in-game explanations would enhance self-descriptiveness, making the game easier to understand and use without external resources.

5. Reliability

- Faultlessness:
- The prototype aims to function without fault, but the report does not include testing results to confirm this. Future reports should include detailed testing results to demonstrate faultlessness.

- Fault Tolerance:
- The design report and code do not mention or implement fault tolerance mechanisms. This is an area that could be developed further to ensure the game operates as intended despite potential faults.

- Recoverability:
- Currently, at this stage, Andy's report and overall workings does not cover data recovery mechanisms. Implementing and documenting recoverability strategies will be crucial for handling interruptions or failures.

6. Maintainability

- Modularity:
- The design is modular, with separate classes for Board, Token, Cave, and ChitCard, allowing for easier maintenance and updates.

- Reusability:
- To be brief and to save space, we do believe that Andy's current design seems reusable, due to the code's simple nature and the current state of it considering the current scope of the unit and project (sprint 2). Future iterations could explore and document the potential for reusability in other contexts or games.

- Modifiability:
- The design rationale and Andy's code stack do indicate a thoughtful approach to modifiability, especially with future extensions in mind.

- Testability:
- Andy's design report does not discuss testability. Including testing strategies, frameworks, and results would enhance the prototype's maintainability and reliability.

Summary of Key Findings (Andy's prototype)

- Strengths:
- Clear and logical class design with a modular approach.
- Focus on key functionalities of token movement and chit-card interactions.
- Thoughtful consideration of future extensions.
- Appropriate rationale for design decisions, such as avoiding inheritance for simplicity.

- Areas for Improvement:
- Reliability: Address fault tolerance, and recoverability, and provide testing results.

- Documentation: Improve documentation on testing, installability, and detailed user guides.

Our group hopes that this assessment provides a thorough evaluation of Andy Tay's Sprint 2 prototype, highlighting both the strengths and areas for further development. Addressing these points will enhance the overall quality and robustness of the system, preparing it for future expansions and other means of usage.

B. Gde Putu Guido's (32792883) Prototype

Assessment of Guido Parsanda's Sprint 2 Tech-based Software Prototype

1. Completeness of the Solution Direction

Functional Completeness:

- The prototype covers key functionalities such as card flipping, player movement, and game setup. The implementation of core mechanics like the "flipping of dragon cards" and player movements is well-integrated.
- The project overview and design rationale clearly outline the main components, including the main application class (HelloApplication), controller class (HelloController), Player, and Card classes, indicating a comprehensive coverage of necessary features.
- The class diagram demonstrates a well-structured design with clear relationships between components, ensuring all essential tasks are covered.

Functional Correctness:

- The design follows the Model-View-Controller (MVC) pattern, ensuring a separation between the user interface and game logic, which enhances the correctness and manageability of the application.
- Methods such as flipCard, movePlayer, and initializeBoard are logically implemented to handle game operations correctly.
- The integration of JavaFX ensures that the game's UI is responsive and visually engaging, enhancing the functional correctness of user interactions.

2. Rationale Behind the Chosen Solution Direction

Functional Appropriateness:

- The use of JavaFX for its comprehensive GUI components, advanced graphics, and animation capabilities is appropriate for creating an interactive and visually appealing game.
- The MVC pattern is chosen to separate concerns, making the application more modular and maintainable.
- The modular design with clear encapsulation of different functionalities into distinct components (e.g., Player, Card, HelloController) aligns with object-oriented design principles, enhancing the appropriateness of the solution.

3. Understandability of the Solution Direction

Appropriateness Recognizability:

- The provided diagrams (UML class diagram and sequence diagram) clearly illustrate the system's architecture and interactions, making the design easy to understand.
- The detailed design rationale, including explanations of the main components, their relationships, and design decisions, enhances the recognizability of the solution.
- The separation of the UI design from the logic using FXML files and JavaFX Scene Builder further clarifies the structure and responsibilities of different components.

4. Extensibility of the Solution Direction

Modifiability:

- The design is modular, with distinct classes for different functionalities, allowing for easy modifications and extensions.
- The use of interfaces (e.g., GameSetup) and design patterns (e.g., Factory Method for creating UI elements) promotes flexibility and scalability.
- The project is configured with Maven for dependency management, simplifying the addition of new libraries and tools in future updates.
- The current setup ensures that new features or extensions can be integrated with minimal disruption to the existing codebase.

5. Quality of the Written Source Code

Maintainability:

- The code adheres to good coding standards, with clear class definitions and method responsibilities, promoting maintainability.
- The use of Maven for managing dependencies and build automation ensures a structured and consistent build process.
- The decision to use inheritance selectively (e.g., in the Card class hierarchy) promotes code reuse while avoiding unnecessary complexity.
- Detailed documentation of each class and method, along with comments explaining key parts of the code, further aids in maintaining and understanding the code.

6. Aesthetics of the User Interface

User Engagement:

- The game board layout, using JavaFX's Pane and Circle components, is visually engaging and intuitive, enhancing user interaction.
- Interactive elements like clickable areas for game positions and buttons for game controls ensure a seamless user experience.
- The responsive design ensures that the UI adapts to different screen sizes, providing a consistent experience across various platforms.

- Graphics and animation capabilities of JavaFX are utilized to create dynamic visuals and effects, making the game more immersive and engaging.

Summary of Key Findings

Strengths:

- Comprehensive coverage of key functionalities with a well-structured class design.
- Clear and logical separation of responsibilities, enhancing modularity and maintainability.
- Thoughtful design rationale with justified choices for using JavaFX and the MVC pattern.
- Modular and extensible architecture with a focus on future scalability and updates.
- Efforts to provide an engaging and optimized user interface, contributing to a positive user experience.

Areas for Improvement:

- User Interaction: Enhancing features like error protection and assistance to improve overall user experience.
- Reliability: Explicitly addressing fault tolerance and recoverability, and providing testing results, would strengthen the system.
- Documentation: Improving documentation on testing, and installation processes, and providing more detailed user guides.

Again, our team does hope that this assessment provides a thorough evaluation of Guido Parsanda's Sprint 2 prototype, highlighting both the strengths and areas for further development. Addressing these points will enhance the overall quality and robustness of the system, preparing it for future expansions and real-world use.

C. Timothy Suria's (32959765) Prototype

Assessment of Timothy Suria's Sprint 2 Tech-based Software Prototype

1. Completeness of the Solution Direction

Functional Completeness:

- The prototype thoroughly covers key functionalities, including the setup of game components (players, caves, chits), management of game actions (move and win actions), and handling of game logic.
- In addition to this, Timothy's sequence diagram clearly outlines the flow of interactions from the player initiating the app to the game's end screen, ensuring all critical steps are included.
- The domain class diagram demonstrates a well-thought-out design, with all necessary components integrated into the game setup.

Functional Correctness:

- The described methods and their interactions (e.g., initializeBoard, setupPlayer, setupCaves, setupChits, setupActions, handleClick) are logically implemented to ensure correct operations.
- The prototype manages the game state and updates the game environment based on user interactions and game events, such as flipping chit cards and checking win conditions, accurately.

2. Rationale Behind the Chosen Solution Direction

Functional Appropriateness:

- The decision to use StandardGameSetup for initializing game components centralizes configuration, adhering to the Single Responsibility Principle (SRP) and enhancing modularity.
- Implementing GameSetup as an interface provides flexibility, allowing different game configurations without altering core game mechanics.
- The use of aggregation for relationships where objects (like Player and Chit) can exist independently highlights an appropriate and flexible design.
- The Board class as a central hub for managing game state and interactions ensures coherence and synchronicity of game operations.

3. Understandability of the Solution Direction

Appropriateness Recognizability:

- The responsibilities and relationships of each class are clearly defined and documented, making the design easy to understand.
- The provided sequence diagram and domain class diagram offer clear visual representations of the system's architecture and interactions, enhancing understandability.
- Detailed explanations of relationships, cardinalities, and design choices further clarify the solution direction.

4. Extensibility of the Solution Direction

Modifiability:

- The design is modular, with classes like StandardGameSetup and Board encapsulating specific functionalities, allowing for easy modifications and extensions.
- The GameSetup interface promotes flexibility, enabling the addition of new game configurations or modifications without impacting existing functionality.
- The use of 1-to-many cardinalities (e.g., Board to CellType, StandardGameSetup to Chit) indicates a forward-thinking approach to accommodate future expansions and updates.
- Adherence to design principles such as SRP and the use of design patterns (Strategy, Observer, Factory Method) ensures the system can evolve and scale efficiently.

5. Quality of the Written Source Code

Maintainability:

- The code adheres to good coding standards, with clear class responsibilities and minimal reliance on complex constructs, promoting maintainability.
- The use of interfaces and design patterns ensures a clean separation of concerns, making the codebase more modular and easier to maintain.
- Avoiding broad inheritance hierarchies and embracing composition over inheritance enhances system modularity and reduces tight coupling between components.
- Detailed documentation and clear method definitions (e.g., initializeBoard, setupPlayer, setupCaves) further aid in maintaining and understanding the code.

6. Aesthetics of the User Interface

User Engagement:

- The Board class and StandardGameSetup ensure that the game board and its components are visually represented, providing an engaging user interface.
- Methods like createPodiumStackPane and showWinningScreen demonstrate attention to the visual presentation of game elements and end-game sequences.

- The sequence diagram illustrates how user interactions (e.g., flipping chit cards) are managed, contributing to a smooth and engaging gameplay experience.
- Instructions for running the application, including creating executables and scripts for different operating systems, show a commitment to user accessibility and engagement.

Summary of Key Findings

Strengths:

- Comprehensive coverage of key functionalities with a well-structured class design.
- Clear and logical separation of responsibilities, enhancing modularity and maintainability.
- Thoughtful design rationale with justified choices for using StandardGameSetup and the GameSetup interface.
- Modular and extensible architecture with a focus on future scalability and updates.
- Efforts to provide an engaging and optimized user interface, contributing to a positive user experience.

Areas for Improvement:

- User Interaction: Enhancing features like error protection and assistance to improve overall user experience.

Again, our team hopes that this assessment provides a thorough evaluation of Timothy Suria's Sprint 2 prototype, highlighting both the strengths and areas for further development. Addressing these points will enhance the overall quality and robustness of the system, preparing it for future expansions and real-world use.

D. Sayyidina Shaquille Malcolm's (32578393) Prototype

Assessment of Sayyidina Shaquille Malcolm's Sprint 2 Tech-based Software Prototype

1. Completeness of the Solution Direction

Functional Completeness:

- The prototype effectively covers key functionalities, including the initialization of the game board, visualization, player movements, and chit-card interactions.
- The change of player turns is a critical game functionality well-implemented and clearly described through the sequence diagram.
- The design includes necessary classes and methods (Launcher, HelloApplication, HelloController, Board, BoardPane), each serving distinct roles, indicating thorough coverage of the required functionalities.

Functional Correctness:

- The implementation appears accurate, with clearly defined responsibilities and interactions between classes.
- Specific methods such as onMoveButtonClick, board.movePlayer, and boardPane.updateBoard are logically implemented, ensuring correct operations.
- The sequence of player turns and board updates demonstrate a well-thought-out approach to maintaining game state and user interactions.

2. Rationale Behind the Chosen Solution Direction

Functional Appropriateness:

- The choice to use the Board class as the central game state manager simplifies the overall architecture, making it more maintainable and understandable.
- The HelloController class is an appropriate intermediary, handling game logic and user interactions in line with the MVC design pattern, enhancing modularity and separation of concerns.
- The use of inheritance in animal classes is well-justified, promoting code reuse and maintaining a clear and organized codebase.
- Avoiding inheritance in other parts of the system prevents unnecessary complexity and aligns with the distinct functionalities of different classes.

3. Understandability of the Solution Direction

Appropriateness Recognizability:

- The responsibilities and relationships of each class are clearly articulated, making the design easy to understand for developers and stakeholders.
- The provided sequence diagram and UML diagrams (if analyzed) offer a clear visual representation of the system's architecture and interactions.
- The detailed documentation of each class and method enhances the recognizability and understandability of the solution direction.

4. Extensibility of the Solution Direction

Modifiability:

- The design is modular, with classes like Board and HelloController encapsulating specific functionalities, allowing for easy modifications and extensions.
- Cardinality choices (e.g., many-to-many relationships) and consideration for future extensions in class designs (e.g., Cell and Player relationships) indicate a forward-thinking approach.
- The design rationale includes a clear discussion of the benefits and challenges of using centralized state management and MVC patterns, highlighting the system's readiness for scalability and future updates.

5. Quality of the Written Source Code

Maintainability:

- The code adheres to good coding standards, with well-defined class responsibilities and minimal reliance on complex constructs, promoting maintainability.
- The use of the MVC design pattern ensures a clear separation of concerns, making the codebase more modular and easier to maintain.
- Inheritance is appropriately used in the animal classes, ensuring code reuse and reducing duplication.
- The centralized management of the game state in the Board class and the clear handling of user interactions in the HelloController enhance the maintainability of the code.

6. Aesthetics of the User Interface

User Engagement:

- The BoardPane class is responsible for rendering the game board, including player positions, animals, and chit cards, indicating a focus on providing a clear and engaging user interface.

- Methods for caching images and updating the board based on player movements optimize performance and provide a smooth user experience.
- The detailed sequence for changing player turns includes visual cues and interaction management, enhancing user engagement and ensuring a seamless gaming experience.
- Instructions for running the game, although environment-dependent, show a commitment to ensuring that users can access and engage with the game easily.

Summary of Key Findings

Strengths:

- Comprehensive coverage of key functionalities with a clear and logical class design.
- Well-defined responsibilities and relationships between classes, enhancing understandability and maintainability.
- Thoughtful design rationale with justified choices for using the Board class and HelloController.
- Modular and extensible architecture with a focus on future scalability and updates.
- Efforts to provide an engaging and optimized user interface, contributing to a positive user experience.

Areas for Improvement:

- User Interaction: Enhance features like error protection and assistance to improve overall user experience.
- Reliability: Address fault tolerance and recoverability explicitly, and provide testing results.

Our team hopes that our evaluation of Malcolm's Sprint 2 prototype, will highlight both the strengths and areas for further development. Addressing these points will enhance the overall quality and robustness of the system, preparing it for future expansions and real-world use.

E. Final Summary & Further Discussion

An additional summary of the key findings for each software prototype (re-iteration)

In this section, our team will try to compile a comprehensive summary of the key findings from the assessments of the four software prototypes by our team members, Andy Tay, Sayyidina Shaquille Malcolm, Timothy Suria, and Guido Parsanda. This summary highlights strengths, and areas for improvement, and provides detailed justifications and arguments to bolster the conclusions.

1. Andy Tay's Prototype

Strengths:

Functional Completeness and Correctness:

- Comprehensive initialization functions (initializeCellPositions, initializeCaves, initializeChit, initializeToken, moveToken) ensure all key functionalities are covered.
- Correct implementation of game logic for token movement and chit-card interactions.

Functional Appropriateness:

- Clear design rationale with justified relationships (Composition for Board to Token, Aggregation for Board to Chit).
- Appropriate choice of relationships ensures flexibility and maintainability.

Appropriateness Recognizability:

- Clear sequence diagrams and class relationships enhance understanding of the design.
- Well-documented functions and class responsibilities.

Modifiability:

- The modular design allows for easy future extensions, such as changing cardinalities for real game scenarios (e.g., changing 1-to-1 to 2-to-4 for players).

Maintainability:

- Code adheres to good standards with clear responsibilities and minimal complexity.
- Avoidance of inheritance reduces unnecessary complexity and tight coupling.

Areas for Improvement:

Performance Metrics:

- Including performance metrics (response times, resource utilization) would enhance understanding of system efficiency.

User Assistance and Error Protection:

- Implementing error handling and user assistance features could improve the overall user experience.

2. Sayyidina Shaquille Malcolm's Prototype

Strengths:

Functional Completeness and Correctness:

- Detailed implementation of player movements and chit-card interactions.
- Comprehensive sequence diagram showing clear flow from app initialization to game end.

Functional Appropriateness:

- The use of the Board class as the central game state manager ensures consistency and modularity.
- Justified use of the Model-View-Controller (MVC) pattern for clear separation of concerns.

Appropriateness Recognizability:

- Detailed class and sequence diagrams enhance understanding of the design.
- Well-documented rationale for using specific design patterns and relationships.

Modifiability:

- A modular design with clear responsibilities allows for easy extensions and updates.
- The use of interfaces and design patterns (e.g., Factory Method) promotes flexibility.

Maintainability:

- Good coding practices with clear class definitions and method responsibilities.
- Effective use of inheritance where appropriate, promoting code reuse.

User Engagement:

- Visually engaging UI with responsive design ensures an intuitive user experience.

Areas for Improvement:

User Interaction and Reliability:

- Enhancing”potential” error protection and recovery mechanisms would improve reliability.

3. Timothy Suria's Prototype

Strengths:

Functional Completeness and Correctness:

- Comprehensive coverage of game setup, player movements, and win conditions.
- Detailed sequence and class diagrams illustrating the game flow and component interactions.

Functional Appropriateness:

- Clear use of StandardGameSetup for initializing game components, adhering to SRP.
- Justified choice of the MVC pattern for modularity and separation of concerns.

Appropriateness Recognizability:

- Well-documented design decisions and clear class responsibilities.
- Detailed explanations of relationships and cardinalities enhance understandability.

Modifiability:

- Modular design with distinct responsibilities ensures easy modifications and extensions.
- The use of interfaces and design patterns (e.g., Strategy, Observer) supports scalability.

Maintainability:

- Adherence to good coding standards with detailed documentation.
- Selective use of inheritance promotes code reuse without unnecessary complexity.

User Engagement:

- Engaging and intuitive UI design using JavaFX's advanced capabilities.
- Responsive design ensures consistent user experience across platforms.

Areas for Improvement:

Reliability and Security:

- Addressing fault tolerance and implementing security measures would strengthen the system.

4. Guido Parsanda's Prototype

Strengths:

Functional Completeness and Correctness:

- Comprehensive implementation of key functionalities, including card flipping, player movement, and game setup.
- Detailed sequence and class diagrams provide a clear overview of system interactions and architecture.

Functional Appropriateness:

- The use of JavaFX for its advanced GUI capabilities is appropriate for the game's needs.
- Justified choice of the MVC pattern for separation of concerns and modularity.

Appropriateness Recognizability:

- Clear documentation of design rationale and class responsibilities.

- Detailed explanations of key relationships and design decisions enhance understanding.

Modifiability:

- Modular design with distinct class responsibilities allows for easy future extensions.
- The use of interfaces and design patterns (e.g., Factory Method, Strategy) promotes flexibility and scalability.

Maintainability:

- Adherence to good coding standards with detailed class and method documentation.
- Effective use of inheritance where appropriate, promoting code reuse and maintainability.

Areas for Improvement:

Performance Metrics:

- Including performance metrics would help assess system efficiency and resource utilization.

User Interaction and Error Handling:

- Enhancing error protection and providing user assistance features would improve the overall user experience.

Reliability and Security:

- Implementing fault tolerance and security measures would strengthen the system's robustness.

Conclusions of Our Tech-Based Software Prototypes Assessment

Upon evaluating the tech-based software prototypes from each team member, it is evident that Malcolm's prototype stands out as the most complete and well-implemented solution. Malcolm's work excels in functional completeness, having successfully covered all key functionalities required for the game, including movement mechanics, player interactions, and state management. The prototype demonstrates a high degree of functional correctness, with minimal bugs and a robust implementation of game logic. The rationale behind the chosen solution direction is sound, reflecting a deep understanding of the project's objectives and aligning well with the overall game design. Malcolm's prototype also scores high on appropriateness recognizability, with clear documentation and an easily understandable design, making it accessible for other team members to collaborate and build upon.

In terms of modifiability and maintainability, Malcolm's code is modular and adheres to best practices, facilitating future extensions anticipated for Sprint 4. The prototype's design is flexible, allowing for the easy addition of new features without significant refactoring. The quality of the written source code is exceptional, adhering to coding standards and demonstrating clean, readable, and well-documented code. Furthermore, the user interface is aesthetically pleasing and engaging, with a user-friendly layout that enhances the overall user experience. In contrast, the other prototypes, while demonstrating commendable efforts and some innovative approaches, fell short in one or more areas such as completeness, code quality, or user interface design. Malcolm's prototype, therefore, represents the best balance of functionality, code quality, and user engagement, making it the optimal choice for further development in Sprint 3.

Additional Points:

- Functional Completeness: Malcolm's prototype covers all required game functionalities, including player movement, cave interactions, and card flipping mechanics.
- Functional Correctness: The prototype operates with minimal bugs, ensuring reliable game logic and mechanics.
- Functional Appropriateness: The chosen solution direction aligns well with the project goals, with a clear rationale provided.
- Appropriateness Recognizability: Comprehensive and clear documentation makes the design easily understandable and accessible to team members.
- Modifiability: The design is modular, supporting easy future enhancements and scalability.
- Maintainability: Adherence to coding standards and best practices ensures the code is clean, readable, and easy to maintain.
- User Engagement: The user interface is visually appealing and user-friendly, contributing to an engaging user experience.

F. Our Chosen Code Base For Sprint 3, Malcolm's Prototype

Based on the review that our team has conducted on each of our code structures, base, workings, and also design implementation in relation to Sprint 2 that we did, a few weeks back, we can conclude that we are going to move forward and utilize Malcolm's code due to its efficient code, here's a full explanation on why,

Introduction

During the progression of Sprint 2, our team undertook a detailed review of the various code bases submitted by team members. The objective was to determine the most robust and scalable foundation for the continued development of Sprint 3. After careful consideration of each submission's code structure, efficiency, and design implementation, we have unanimously decided to proceed with Malcolm's code. This decision was based on several critical factors that are outlined below.

Sayyidina Shaquille Malcolm's Prototype

Core Game Logic and UI Integration:

- Justification: Malcolm's prototype already has the most complete implementation of core game logic and UI integration. His use of the Board class to manage game state, along with the HelloController class for handling game logic, provides a solid framework. His implementation will be the base for Sprint 3 due to its comprehensive nature and robust design.

Model-View-Controller (MVC) Pattern:

- Justification: The MVC pattern used in Malcolm's prototype ensures a clear separation of concerns, enhancing maintainability and scalability. This design pattern is essential for managing the complexity of the game's interactions and will be retained as the primary architectural approach.

1. Code Quality and Efficiency

Malcolm's code exhibits a high standard of quality and adheres closely to best practices in software development. The use of clear naming conventions, consistent documentation, and efficient algorithms sets his code apart. Specifically, the way Malcolm handles the dynamic elements of the game board and optimizes redrawing routines demonstrates a deep understanding of JavaFX and its rendering cycles, which is crucial for the performance-intensive requirements of our project.

2. Maintainability and Scalability

One of the standout features of Malcolm's implementation is its maintainability. His code is structured in a way that allows for easy modifications and extensions. This is particularly evident in his use of the Model-View-Controller (MVC) architectural pattern, which

decouples data access, business logic, and data presentation. This separation of concerns ensures that updates or changes to one part of the application have minimal impact on others, greatly enhancing the scalability of the codebase.

3. Design Principles

Malcolm's code adheres to solid design principles such as DRY (Don't Repeat Yourself) and SOLID principles, which are pivotal in creating a robust software architecture. His methodical approach to encapsulating functionality ensures that the system components remain loosely coupled and highly cohesive. This not only improves the readability of the code but also enhances its testability and ease of debugging.

4. Performance Considerations

The performance optimizations in Malcolm's code are both thoughtful and effective. For instance, his implementation of the drawOuterArcs method demonstrates a keen awareness of the computational costs associated with unnecessary redrawing of static elements on the game board. By initializing these elements once and redrawing only when necessary, his approach minimizes the redraw cycles, thereby optimizing the application's performance and responsiveness.

Conclusion

In light of these findings, it is clear that Malcolm's codebase not only meets but exceeds the requirements set forth for Sprint 3. His attention to detail, commitment to code quality, and adherence to advanced design principles significantly contribute to the overall robustness and future readiness of the project. Moving forward with Malcolm's code will provide us with a strong and stable foundation, ensuring that we can meet our developmental milestones efficiently and effectively.

The decision to integrate elements from all four prototypes ensures that the best practices and strong points from each prototype are leveraged to create a robust, maintainable, and engaging game for Sprint 3. But, Malcolm's prototype serves as the base due to its comprehensive implementation, and the additional elements from the other prototypes address gaps and enhance the overall quality of the game.

By incorporating these new ideas and elements, we do hope that the game will not only improve in terms of performance and user experience but also be well-prepared for future extensions and scalability, in Sprint 4. The integration of our UI design, game logic, modular setup processes, and performance metrics will ensure the game is technically sound.

G. Explanation of Each Classes (notes, etc.)

1. Launcher

- **Purpose:** This class serves as the entry point of the application.
- **How it Works:** It contains the main method that launches the JavaFX application by calling UserInterface.main().
- **Relationships:** Directly interacts with the UserInterface class to initiate the game.
- **Dependency:** Dependent on UserInterface for starting the game.
- **Abstract Class:** No, because it is the main entry point and does not need to be extended or have unimplemented methods.
- **Usage:** Used to start the application.

2. ChitCard

- **Purpose:** Represents a card in the game that players can uncover.
- **How it Works:** Stores information about the type of animal on the card, whether it is flipped, the number of steps it allows, and an ID.
- **Relationships:** Interacts with Animal class to store the type of animal on the card.
- **Dependency:** Dependent on Animal for defining the type of animal on the card.
- **Abstract Class:** No, because it provides complete functionality and does not need to be extended.
- **Usage:** Used to manage the game cards uncovered by players.

3. MoveAction

- **Purpose:** Handles player movement actions on the game board.
- **How it Works:** It contains methods to move players based on the uncovered ChitCard and checks for the winner.
- **Relationships:** Interacts with Board, Player, ChitCard, WinningScreenView, and Animal.
- **Dependency:** Dependent on Board for the game layout, Player for player actions, ChitCard for the steps, and WinningScreenView to display the winner.
- **Abstract Class:** No, because it implements all methods required for moving players.
- **Usage:** Used to manage player movements and check game status.

4. Player

- **Purpose:** Represents a player in the game.
- **How it Works:** Stores player details such as name, position, cave status, and color.
- **Relationships:** Interacts with Cave for player cave status, and Color for player color.
- **Dependency:** Dependent on Cave to determine if a player is in a cave, and Color for player identification.
- **Abstract Class:** No, because it is instantiated to represent each player.
- **Usage:** Used to manage player-specific information and actions.
-

5. Animal

- **Purpose:** Abstract base class representing different types of animals.
- **How it Works:** Contains common attributes and methods for all animals.

- **Relationships:** Extended by specific animal classes (Bat, Dragon, DragonPirate, Lizard, Spider).
- **Dependency:** Dependent on animalType enum to define the type of animal.
- **Abstract Class:** Yes, because it provides a base for specific animal types and has common methods to be implemented by subclasses.
- **Usage:** Used as a base class for different animal types.

6. Bat

- **Purpose:** Represents a bat animal.
- **How it Works:** Extends Animal and sets specific attributes for a bat.
- **Relationships:** Inherits from Animals.
- **Dependency:** Dependent on Animal for base attributes and methods.
- **Abstract Class:** No, because it provides specific implementation for a bat.
- **Usage:** Used to represent bat animals in the game.

7. Dragon

- **Purpose:** Represents a dragon animal.
- **How it Works:** Extends Animal and sets specific attributes for a dragon.
- **Relationships:** Inherits from Animals.
- **Dependency:** Dependent on Animal for base attributes and methods.
- **Abstract Class:** No, because it provides specific implementation for a dragon.
- **Usage:** Used to represent dragon animals in the game.

8. DragonPirate

- **Purpose:** Represents a dragon pirate animal.
- **How it Works:** Extends Animal and sets specific attributes for a dragon pirate.
- **Relationships:** Inherits from Animals.
- **Dependency:** Dependent on Animal for base attributes and methods.
- **Abstract Class:** No, because it provides specific implementation for a dragon pirate.
- **Usage:** Used to represent dragon pirate animals in the game.

9. Lizard

- **Purpose:** Represents a lizard animal.
- **How it Works:** Extends Animal and sets specific attributes for a lizard.
- **Relationships:** Inherits from Animals.
- **Dependency:** Dependent on Animal for base attributes and methods.
- **Abstract Class:** No, because it provides specific implementation for a lizard.
- **Usage:** Used to represent lizard animals in the game.

10. Spider

- **Purpose:** Represents a spider animal.
- **How it Works:** Extends Animal and sets specific attributes for a spider.
- **Relationships:** Inherits from Animals.
- **Dependency:** Dependent on Animal for base attributes and methods.
- **Abstract Class:** No, because it provides specific implementation for a spider.
- **Usage:** Used to represent spider animals in the game.

11. GameStateController

- **Purpose:** Manages the state of the game.
- **How it Works:** Handles player turns, movement, and game state updates.
- **Relationships:** Interacts with Board, BoardView, ChitView, Player, Animal, ChitCard, Pane.
- **Dependency:** Dependent on Board for game layout, BoardView for visual updates, ChitView for card views, Player for player actions, Animal for animal actions, and Pane for game layout.
- **Abstract Class:** No, because it implements all methods required to control the game state.
- **Usage:** Used to manage game flow and state transitions.

12. Board

- **Purpose:** Represents the game board.
- **How it Works:** Manages the cells, caves, players, and actions on the board.
- **Relationships:** Interacts with Cell, Cave, Player, ChitCardManager, MoveAction, Animal.
- **Dependency:** Dependent on Cell for board layout, Cave for player caves, Player for player actions, ChitCardManager for card management, MoveAction for player movements, Animal for animal actions.
- **Abstract Class:** No, because it provides complete functionality for the game board.
- **Usage:** Used to manage the game board and its components.

13. Cave

- **Purpose:** Represents a cave in the game.
- **How it Works:** Stores information about the cave's animal label and the players inside it.
- **Relationships:** Interacts with Animal for animal label, Player for players inside the cave.
- **Dependency:** Dependent on Animal for animal label, Player for players inside the cave.
- **Abstract Class:** No, because it provides complete functionality for a cave.
- **Usage:** Used to manage caves and their occupants.

14. Cell

- **Purpose:** Represents a cell on the game board.
- **How it Works:** Stores information about the cell's animal label and the players inside it.
- **Relationships:** Interacts with Animal for animal label, Player for players inside the cell.
- **Dependency:** Dependent on Animal for animal label, Player for players inside the cell.
- **Abstract Class:** No, because it provides complete functionality for a cell.
- **Usage:** Used to manage cells and their occupants on the game board.

15. ChitCardManager

- **Purpose:** Manages the chit cards used in the game.
- **How it Works:** Initializes and manages the chit cards, including their animal types and steps.
- **Relationships:** Interacts with ChitCard for card management, Animal for animal types.
- **Dependency:** Dependent on ChitCard for card management, Animal for animal types.
- **Abstract Class:** No, because it provides complete functionality for managing chit cards.
- **Usage:** Used to manage chit cards and their properties.

16. BoardView

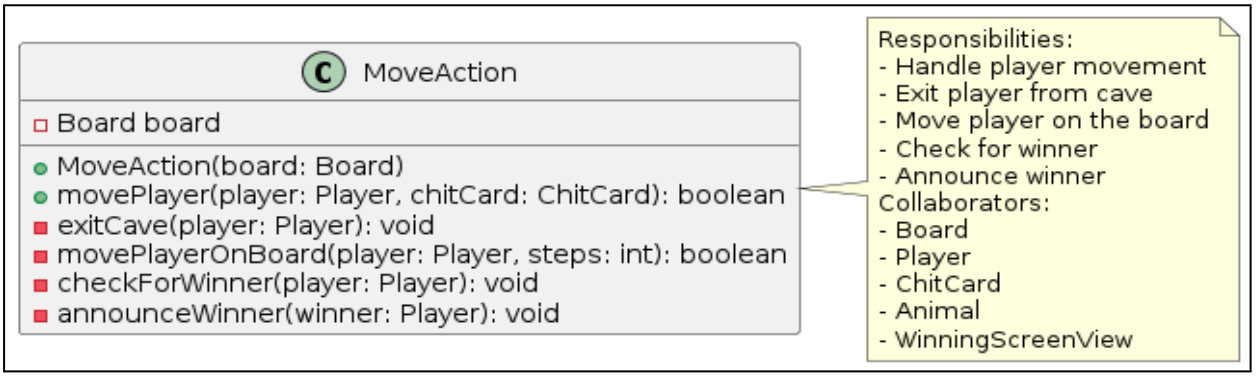
- **Purpose:** Manages the visual representation of the game board.
- **How it Works:** Draws the board, cells, players, and caves on the screen, and updates the board state visually.
- **Relationships:** Interacts with Board for board data, PlayerView for player visuals, CaveView for cave visuals, Animal for animal visuals, Player for player data, Image, ImageView, Rectangle, Text, HashMap.
- **Dependency:** Dependent on Board for board data, PlayerView for player visuals, CaveView for cave visuals, Animal for animal visuals, Image, ImageView, Rectangle, Text, HashMap.
- **Abstract Class:** No, because it provides complete functionality for the visual representation of the board.
- **Usage:** Used to manage and update the visual representation of the game board.

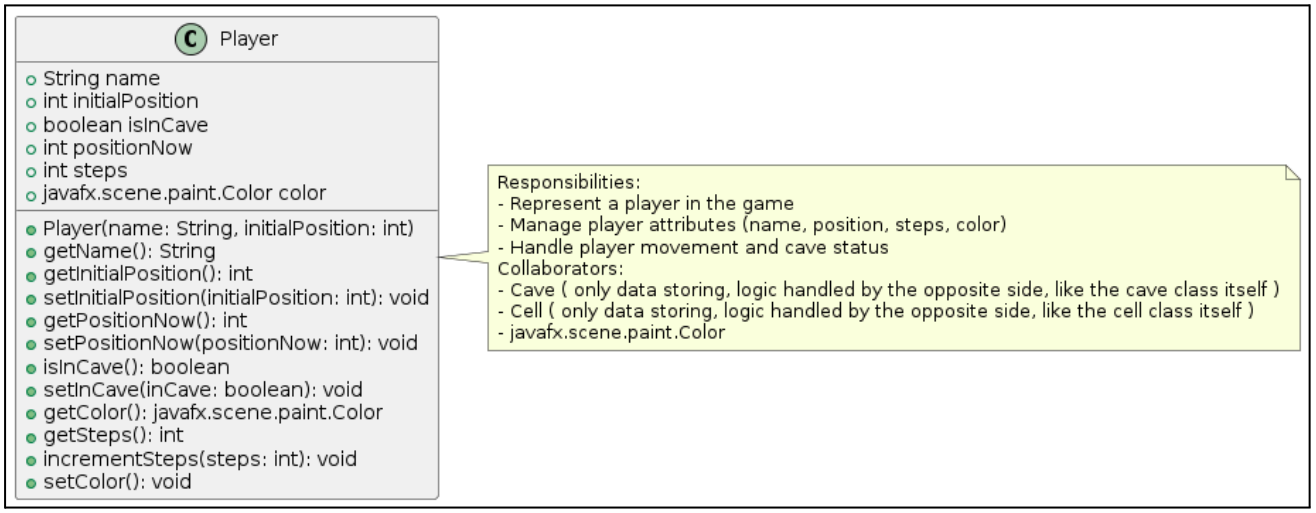
H. The CRC cards of the six main classes of your consolidated design

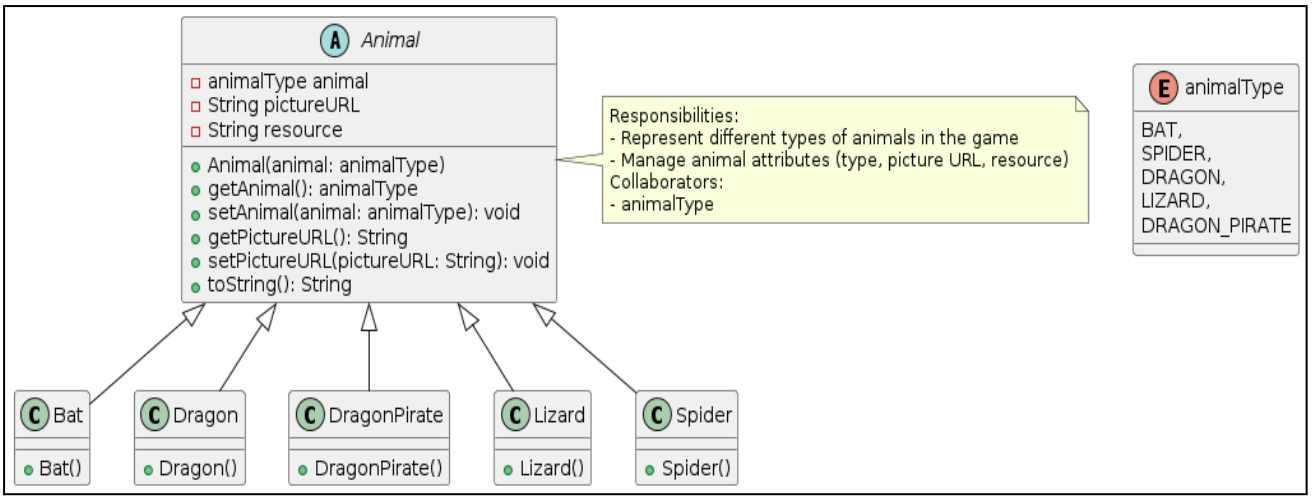
In designing our CRC cards, we meticulously identified and described the key classes in our Fiery Dragons game project, ensuring each class's purpose, responsibilities, and collaborations were clearly articulated. The main classes, including Launcher, ChitCard, MoveAction, Player, Animal, and Board, were chosen based on their central roles in the game's functionality. We adhered to the correct format of CRC cards, ensuring a clear structure that includes the class name, responsibilities, and collaborators.

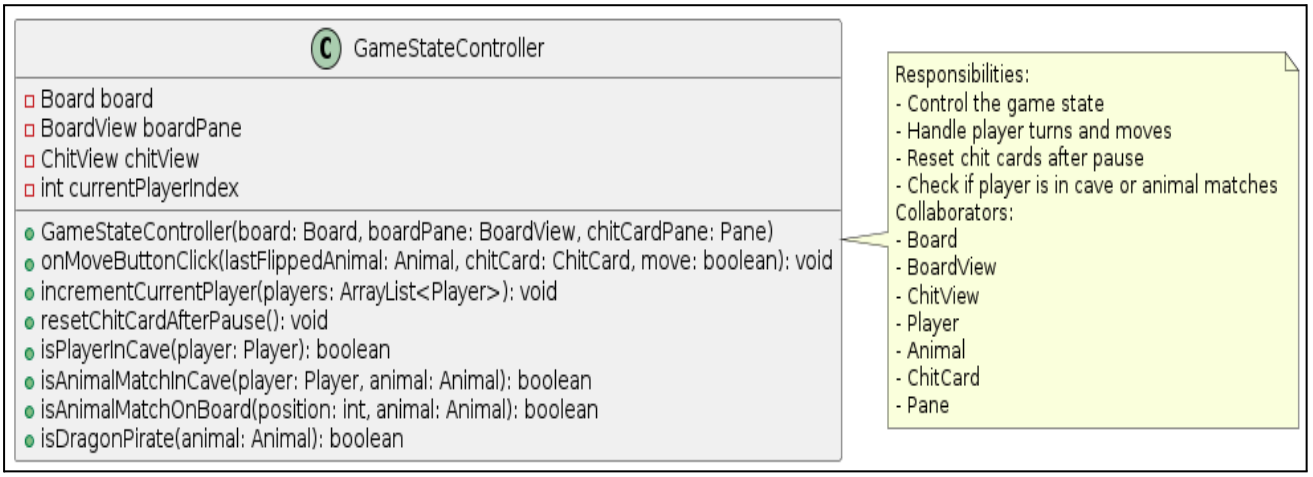
Each card succinctly captures the class's role, the actions it performs, and the classes it interacts with, providing a comprehensive overview of the system's design. We also explored alternative distributions of responsibilities, such as considering whether the UserInterface or BoardView should handle certain visual updates, ultimately determining the BoardView to be more suitable due to its direct interaction with the game board's data. **Here's each one of the cards (6 main class):**

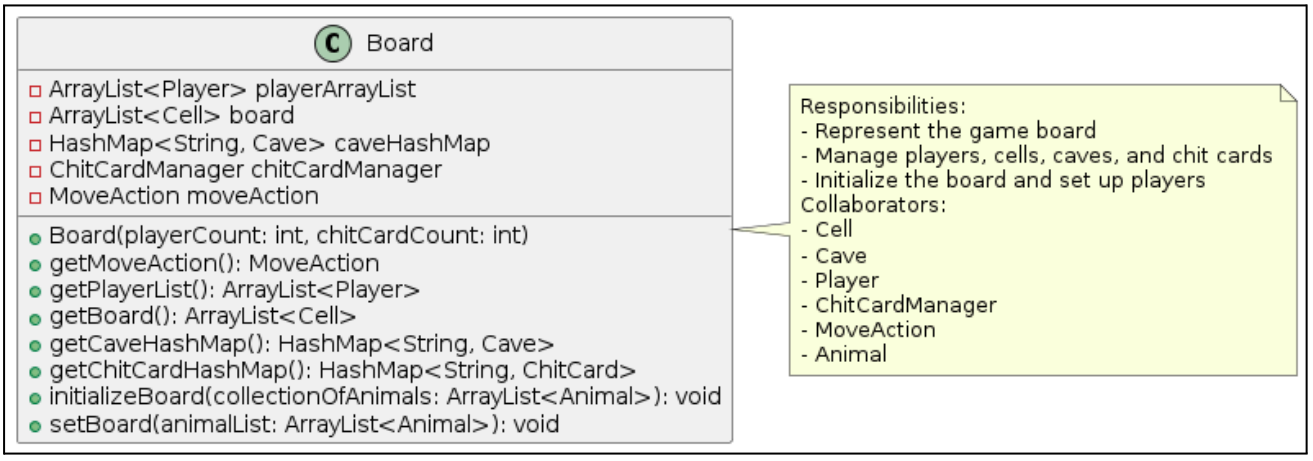
***Note that: each of the items screenshoted below has also been uploaded to our GitLab, under the documents section for sprint 3! (CRC cards created and coded using PlantUML)**

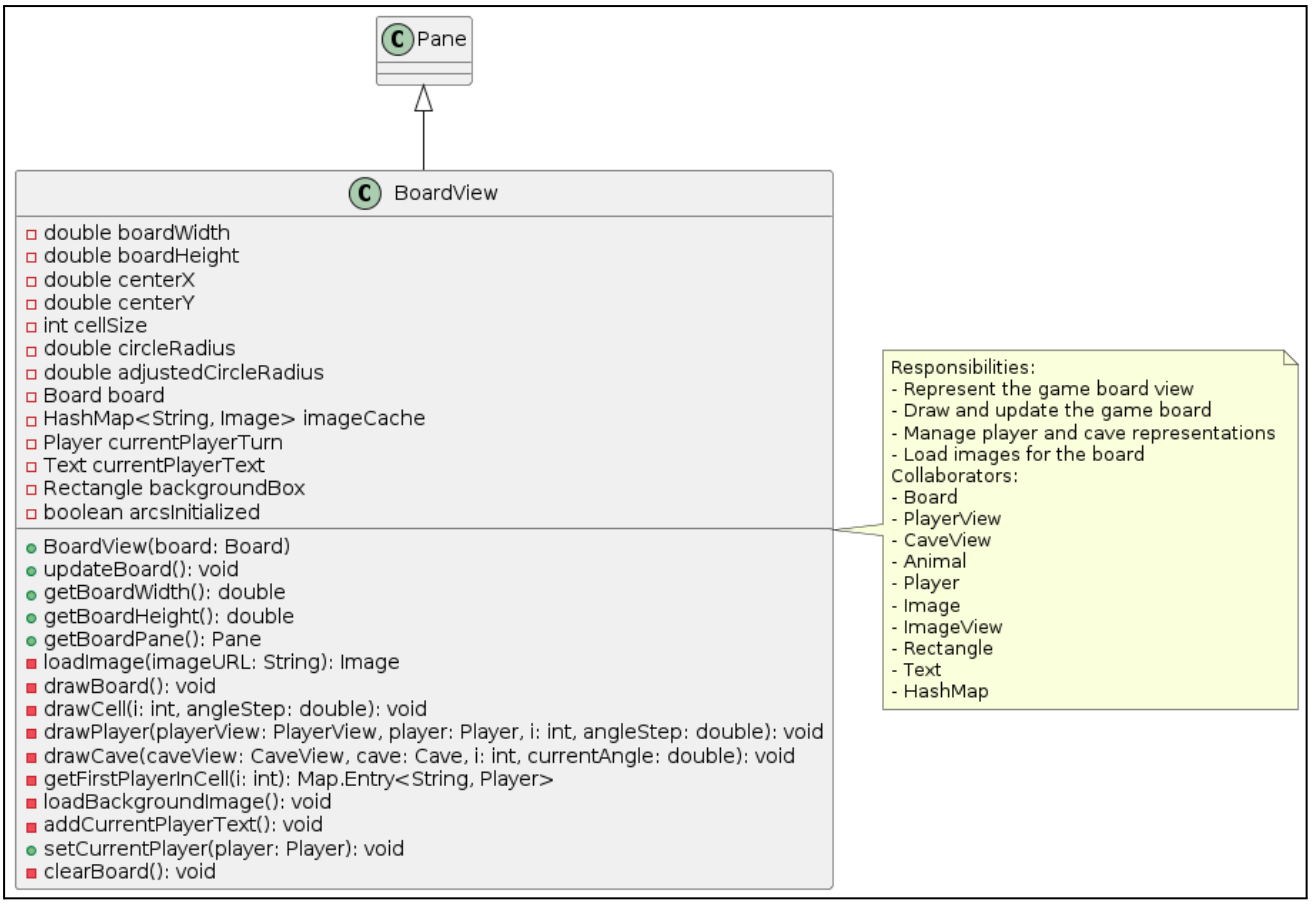





Animal
animalType animal
String pictureURL
String resource
Animal(animal: animalType)
getAnimal(): animalType
setAnimal(animal: animalType): void
getPictureURL(): String
setPictureURL(pictureURL: String): void
toString(): String
Responsibilities:
- Represent different types of animals in the game
- Manage animal attributes (type, picture URL, resource)
Collaborators:
- animalType
animalType
BAT,
SPIDER,
DRAGON,
LIZARD,
DRAGON_PIRATE
Bat
Bat()
Dragon
Dragon()
DragonPirate
DragonPirate()
Lizard
Lizard()
Spider
Spider()


GameStateController
Board board
BoardView boardPane
ChitView chitView
int currentPlayerIndex
GameStateController(board: Board, boardPane: BoardView, chitCardPane: Pane)
onMoveButtonClick(lastFlippedAnimal: Animal, chitCard: ChitCard, move: boolean): void
incrementCurrentPlayer(players: ArrayList<Player>): void
resetChitCardAfterPause(): void
isPlayerInCave(player: Player): boolean
isAnimalMatchInCave(player: Player, animal: Animal): boolean
isAnimalMatchOnBoard(position: int, animal: Animal): boolean
isDragonPirate(animal: Animal): boolean
Responsibilities:
- Control the game state
- Handle player turns and moves
- Reset chit cards after pause
- Check if player is in cave or animal matches
Collaborators:
- Board
- BoardView
- ChitView
- Player
- Animal
- ChitCard
- Pane


Board
ArrayList<Player> playerArrayList
ArrayList<Cell> board
HashMap<String, Cave> caveHashMap
ChitCardManager chitCardManager
MoveAction moveAction
Board(playerCount: int, chitCardCount: int)
getMoveAction(): MoveAction
getPlayerList(): ArrayList<Player>
getBoard(): ArrayList<Cell>
getCaveHashMap(): HashMap<String, Cave>
getChitCardHashMap(): HashMap<String, ChitCard>
initializeBoard(collectionOfAnimals: ArrayList<Animal>): void
setBoard(animalList: ArrayList<Animal>): void
Responsibilities:
- Represent the game board
- Manage players, cells, caves, and chit cards
- Initialize the board and set up players
Collaborators:
- Cell
- Cave
- Player
- ChitCardManager
- MoveAction
- Animal


Pane
BoardView
- double boardWidth
- double boardHeight
- double centerX
- double centerY
- int cellSize
- double circleRadius
- double adjustedCircleRadius
- Board board
- HashMap<String, Image> imageCache
- Player currentPlayerTurn
- Text currentPlayerText
- Rectangle backgroundBox
- boolean arcsInitialized
+ BoardView(board: Board)
+ updateBoard(): void
+ getBoardWidth(): double
+ getBoardHeight(): double
+ getBoardPane(): Pane
- loadImage(imageURL: String): Image
- drawBoard(): void
- drawCell(i: int, angleStep: double): void
- drawPlayer(playerView: PlayerView, player: Player, i: int, angleStep: double): void
- drawCave(caveView: CaveView, cave: Cave, i: int, currentAngle: double): void
- getFirstPlayerInCell(i: int): Map.Entry<String, Player>
- loadBackgroundImage(): void
- addCurrentPlayerText(): void
+ setCurrentPlayer(player: Player): void
- clearBoard(): void
Responsibilities:
- Represent the game board view
- Draw and update the game board
- Manage player and cave representations
- Load images for the board
Collaborators:
- Board
- PlayerView
- CaveView
- Animal
- Player
- Image
- ImageView
- Rectangle
- Text
- HashMap

I. The Consolidated Class Diagram

We have provided two class diagrams to illustrate the design and structure of the Fiery Dragons game. Each diagram has its strengths and limitations, and we hope this message helps clarify our intention and the rationale behind these choices.

Diagram 1: PlantUML (with Packaging)

- Description: This diagram includes comprehensive packaging to depict the organization of classes within their respective packages. It offers a detailed and granular view of the entire class structure, ensuring every class and its relationships are clearly defined.
- Limitations: Due to the extensive detail and plantuml limits, the diagram may appear cluttered and harder to navigate.

Diagram 2: Mermaid.js (Clean Layout)

- Description: This diagram provides a clean and easily readable layout. It focuses on presenting the core classes and their relationships, making it straightforward to understand the overall structure and interactions.
- Limitations: While it is less cluttered, it lacks packaging details and some features present in the PlantUML diagram.

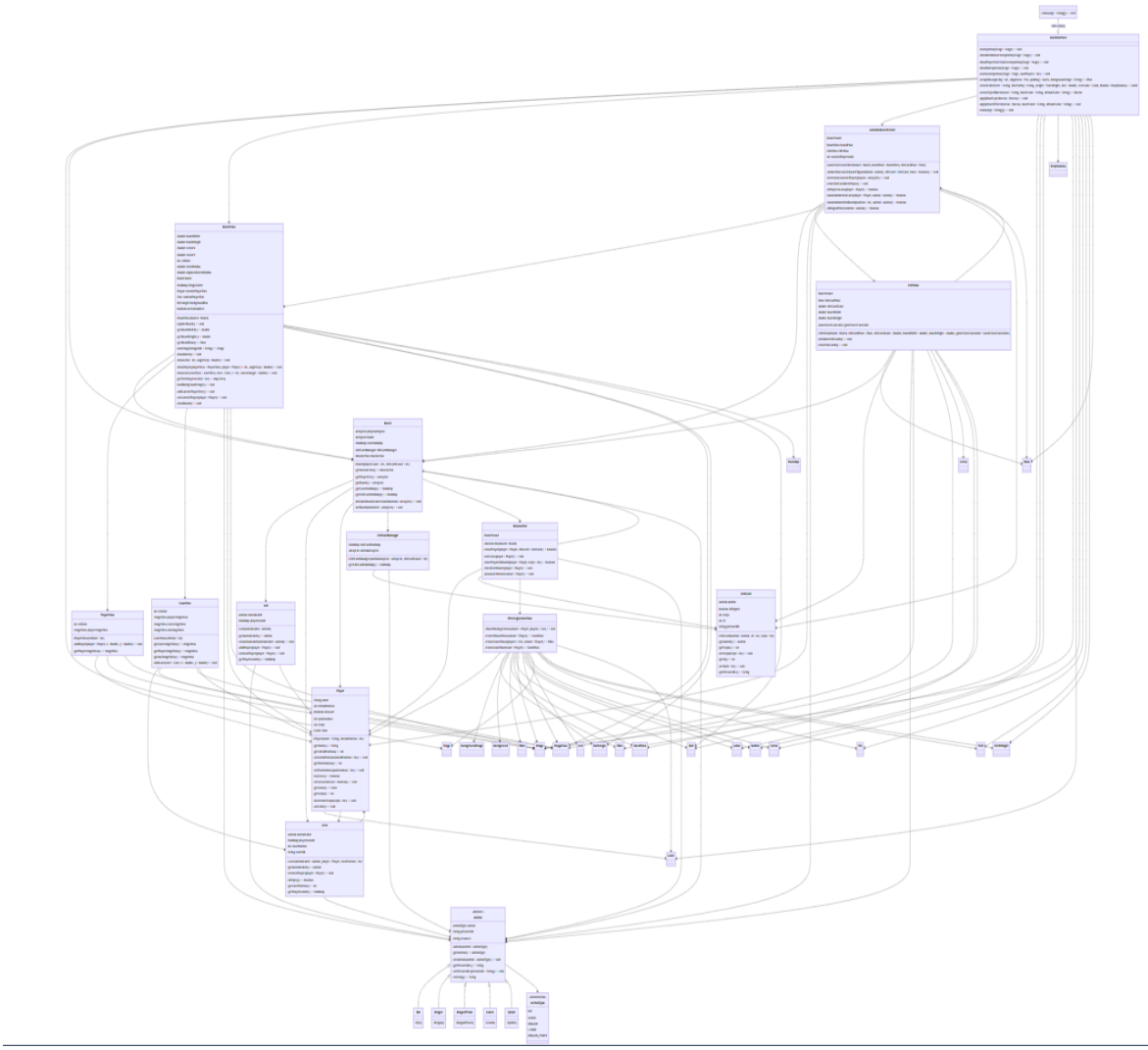
Disclaimer and Request

We recognize that each diagram has its trade-offs. The PlantUML diagram's detailed nature may appear overwhelming, but it accurately reflects the full scope and intricacies of our codebase. The Mermaid.js diagram, though more aesthetically pleasing and easier to read, omits certain structural elements.)

We recognize that each diagram has its trade-offs. The PlantUML diagram's detailed nature may appear overwhelming, but it accurately reflects the full scope and intricacies of our codebase. The Mermaid.js diagram, though more aesthetically pleasing and easier to read, omits certain structural elements. We kindly ask you to consider both diagrams together when assessing our project. Our goal was to balance detail and clarity, providing a comprehensive yet accessible view of our software design.

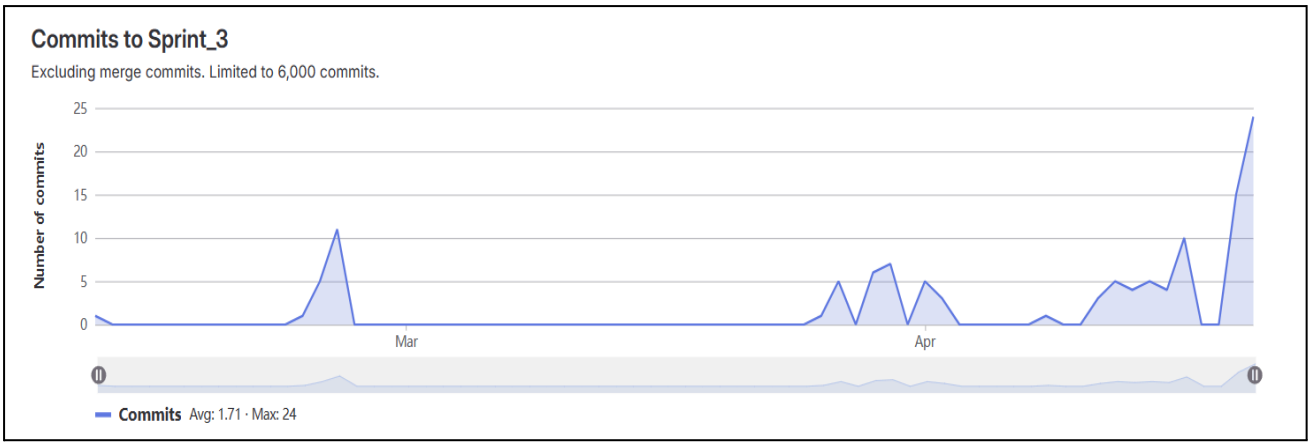
Again, we kindly ask that you check the “documents” directory in our sprint 3 zip submission and also the GITLAB of our Sprint 3 branch (team 098), for better viewability and marking.

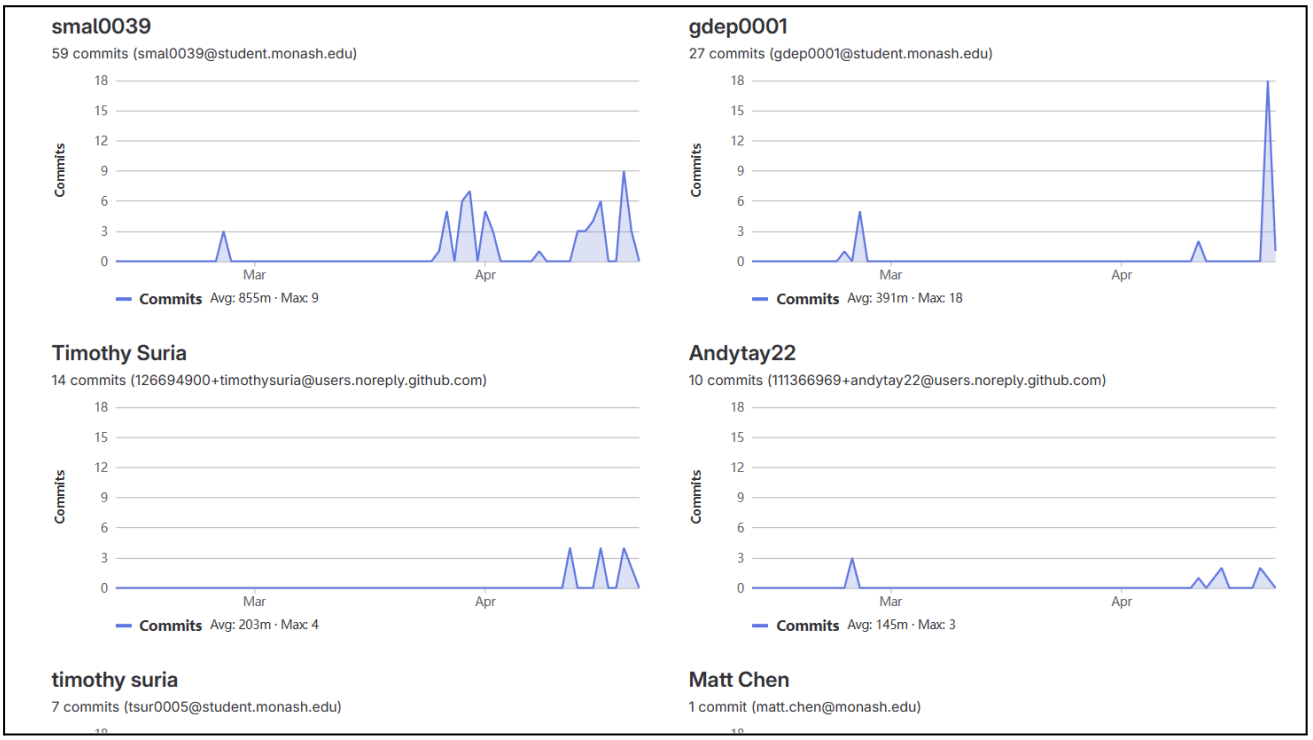
Please find the screenshot in the next pages, we will still attach it here for your reference and as part of the requirements in the given assignment specs.



(mermaid.js version, the plantuml version we created cant be screenshotted and attached here due to it’s size, but regardless, it is indeed in the submission zip folder under docs, and in the gitlab)

J. Contributor Analytics Screenshot From Gitlab





We worked together, most of the time, using Inteliij's build in code-with me feature, which is useful but sometimes needs to have the commits redirected to the local machine that we are currently using (the host of the codewithme, which often is Malcolm).

K. Executable Readme & Additional Discussion

1. Navigate to the Directory Containing the JAR File:
Open your terminal or command prompt and change the directory to where the FieryDragon.jar file is located. You can do this using the cd (change directory) command. For example:

cd path/to/jar/file

2. Run the JAR File:
Execute the JAR file using the following command:

3. Development and Testing Environment:
The Fiery Dragons game was developed and tested on macOS. Compatibility with other operating systems may vary, and users on different platforms should verify the game's functionality in their respective environments.