Q. describe your overall approach to implementing the game, – state and justify the adjustments and modifications to the initial design of the project (shown in class diagrams and use cases from Phase 1), – explain the management process of this phase and the division of roles and responsibilities, – list external libraries you used, for instance for the GUI and briefly justify the reason(s) for choosing the libraries, – describe the measures you took to enhance the quality of your code, – and discuss the biggest challenges you faced during this phase.

Adjustment and Modifications: Added Threads (attach updated UML and modification made)

1. **Threads for Improved Performance**: One major addition was the use of threads to manage various game elements asynchronously. For example, enemy movements, animations, and player actions are handled on separate threads, allowing for smoother gameplay and more responsive interactions. This modification required an update to the UML to reflect the addition of a Thread component attached to game objects that require asynchronous behaviour.
2. **Enhanced Enemy Behavior**: We initially planned simple behaviour for enemy objects, but later decided to introduce more complex movement patterns and decision-making processes. This required additional methods in the enemy classes and an update to the class diagrams.
3. **Improved State Management**: We refactored our state management to ensure consistency across different game states. This involved modifying the GameState class and ensuring that transitions between states (e.g., from "Game Over" to "Restart") were smooth and did not require additional resources.

Management Process:

1. **Rendered and Threads Developer:** Produces rendering on object for Panel and Window and
2. **Enemy, Movement and Object Classes**: Focused on enemy AI and object classes
3. **Core Game Mechanics and UI/UX Developer:** Manages calling all appropriate methods, managing GameState and all graphical elements, interface design, and user interactions.

Libraries Used:

1. **javax.swing**: We used the javax.swing library for the GUI. Swing provides a robust toolkit for building desktop applications with Java, and it offers sufficient functionality to implement interactive game elements like buttons, panels, and custom graphics. Swing was chosen due to its flexibility in rendering 2D graphics, making it ideal for our game's interface and visual elements.
    1.1. JPanel: Used for creating the main game panel, handling the drawing of game elements and game loop updates.
    1.2. JFrame: Used as the main game window to hold and display the game panel and other GUI components.
    1.3. JButton: Used for interactive buttons within menus and UI elements in the game.
    1.4. JLabel: Used for displaying text and icons, such as scores, messages, and images within the game.
    1.5. Timer: Utilised for handling timed events, like animations and enemy movements, enabling smooth and controlled updates.

2. **Java Concurrency Package**: The java.util.concurrent package was used to implement multithreading. This allowed us to manage real-time aspects of the game, such as enemy movement and animations, without interrupting the main game loop. Threads improved responsiveness and helped achieve smoother gameplay.
    2.1. ExecutorService: Used to manage thread pools, which allow us to control multiple concurrent tasks for handling game events in parallel.
    2.2. ScheduledExecutorService: Used to schedule periodic tasks, such as updating enemy positions, checking collisions, and other time-based game mechanics.
    2.3. Thread: Directly used in some cases to handle specific tasks where precise control over execution was needed, such as animating complex behaviours.

3. **JUnit**: We utilised JUnit for automated testing. By creating test cases for core functionalities, like player movements and collision detection, JUnit enabled us to ensure the reliability and correctness of key game mechanics early in development.

3.1. assertEquals and other assertions: Used to verify expected outcomes for core game functionalities like player movement, score calculations, and collision detection.
3.2. @Before and @After annotations: Used to set up and tear down test environments before and after each test, ensuring isolated and reliable test results.
3.3. @Test: Used to mark methods as test cases, making it easy to identify and run specific tests focused on individual game mechanics.

Measures to enhance code