

GROUP 11
Nicholas Lagasse
Ekansh Chawla
Felmer Macadangdang

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 I), – 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.

Overall approach: Our approach was to split the project up into three different jobs as one of our teammates dropped the course. For the midway-progress deadline, we managed to get the game window open, draw some of the sprites to the window, and move a player character around. We also managed to make most of our classes by the halfway point, but did not fully implement them at that point. We ended up having to give all team members more work because of the fourth member leaving, which led us to remove some of the implementations we had planned such as the special powerup allowing the player to jump over any holes.

Adjustment and Modifications:

1. **Threads for Improved Performance:** One major addition was the use of a thread to manage various game elements at the same time. The thread library allowed us to sequentially call functions we had made, and for the game logic to run well sequentially. We had initially planned to do multithreaded programming, but since most of the time complexity of the project calculations did not take more than a few milliseconds, we decided to make the project work sequentially.
2. **Game states:** We ended up removing the concept of game states such as start screen, end screen, and game screen, as it was not necessary for the base game to work, and we ran out of time for the implementation of it.
3. **Remove most UI elements:** We initially wanted to have a start and end screen, but due to lack of time and resources we had to scrap the idea. This meant that all of the UI classes, buttons, and difficulty selectors ended up getting removed from the project. It was a good decision because we needed to put all of our focus into just getting the game to run, enemies to spawn, and player score to print properly.

Management Process:

We initially had the project split up more evenly into four roles, but since our fourth member left we were forced to give every member more work. We tried our best to have weekly meetings to see what everyone had worked on, but found it easier to work in voice calls with each other as communicating ideas became much easier over voice. We split up all three team members into different sections of the project as follows:

1. **Window, Render, Main, Keyboard Input, and Threads:** Produces the window of the game, rendering all game data, as well as dealing with keyboard input and all thread function calls (Nicholas Lagasse)
2. **Enemies, Items, and Object Classes:** Focused on enemy AI and object classes, and object spawning, as well as updating main with all classes (Ekansh Chawla)
3. **UI and Maze Generation:** Created game sprites and maze wall generation (Felmer Macadangdang)

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.
2. **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.
 - 2.1. **assertEquals and other assertions:** Used to verify expected outcomes for core game functionalities like player movement, score calculations, and collision detection.
 - 2.2. **@Before and @After annotations:** Used to set up and tear down test environments before and after each test, ensuring isolated and reliable test results.
 - 2.3. **@Test:** Used to mark methods as test cases, making it easy to identify and run specific tests focused on individual game mechanics.
3. **java.awt Colour, Dimension, Graphics, Image, ImageIO, IOException:** We used the java.awt libraries for rendering objects and the game window, as well as image importing and asserting.
 - 3.1. **Colour:** Used for generic RGB colours
 - 3.2. **Dimension:** Used for window size
 - 3.3. **Graphics:** Used for graphics objects that render and paint images
 - 3.4. **Image:** Used for holding png data
 - 3.5. **ImageIO:** Used for reading sprites from the sprite folder
 - 3.6. **IOException:** Used for file not found exceptions
4. **Runnable:** We implemented the 'Runnable' class in BaseThread to make a single thread to run the main code sequentially.
5. **ArrayList:** We used arraylist for most of our collections of objects, as it had a lot of useful methods such as size().

Measures to enhance code

1. **Object Factories:** We implemented object factories in our code from the lecture, as most of our objects such as item, enemy, and player extended gameObject, which would make it easier for us to create everything. A lot of the planning from phase 1 helped a lot, as we managed to create proper extension of parent classes, which made the code a lot more readable and manageable.
2. **Getters and Setters:** We had all of our code held in the same folder, so using public getters and setters across classes was easier than if we had separated related classes into different subfolders. The only subfolder we had present was for sprites, which made more sense to us.

Biggest challenges

1. **3 Group Members:** Having only three group members made the project much harder than anticipated. It took us longer to complete the code and to ensure that each group member had a fair allocation of tasks. With an additional group member, we could have implemented more of the features we initially planned, but with three members, we focused on meeting the base code requirements. We tried our best to manage with the resources we had, ultimately getting a working base version.
2. **Streamlining Classes:** Due to time constraints, we chose to simplify the project structure by removing some redundant classes, such as Cell and CellArray, which added extra complexity without significantly enhancing functionality. This allowed us to focus on implementing core requirements instead of maintaining multiple interconnected classes. We streamlined our game logic by working directly with a GameObject array, which simplified development and reduced unnecessary dependencies between classes.
3. **Integration Challenges:** Although we used GitHub for collaboration, the interconnected nature of our code meant that one team member often had to wait for another to finish their portion before updating their own code. We addressed this by scheduling coding sessions together over voice calls, which allowed us to troubleshoot in real time, stay on the same page, and streamline the merging process.
4. **Time Constraints and UI Development:** Due to time constraints, we prioritized the core functionality and had limited opportunity to work on the user interface components such as MainWindow and other UI elements. These would have enhanced the project's usability and polish, but we focused on completing the base functionality given the time and resources available.
5. **Addition of BaseThread:** To manage the game loop, we implemented a BaseThread class, which helped in handling the main game updates and rendered the game at a consistent interval. This addition was crucial for making the gameplay smooth and responsive, allowing us to handle enemy and player movements within a set time cycle.