PHASE 2 REPORT

Overall Approach:

Our team adopted a dynamic approach to project management and implementation. We regularly convened, both in-person and via calls, to distribute tasks based on our individual strengths. Our initial strategy was to build upon our Phase 1 UML and Use Cases. However, we soon encountered complexities in translating these designs into a functional game. For instance, when creating the Position class, we realized the need for more sophisticated handling of game grid dynamics, leading to an enhanced algorithm for movement and collision detection. This practical experience was invaluable in understanding the nuances of game development. At first (before facing the issues, which will be later explained), we did this approach:

1- Made the base for our game. For instance, we made a class for the user inputs, levels, main class, objects (barriers, enemies, player, traps, etc.).
2- We made a Position class that represented a position in the game grid with X and Y coordinates.
3- We also made frontend classes for our GUI.

However, these classes were mainly empty, and we decided to complete them after making the whole skeleton and base of the game. This was our plan for the midway deadline; to have all the base (with some simple implementations for some of them), so we could finish the game by the actual deadline. However, we were facing some confusions and issues, so we had to make changes to our initial plans in Phase 1. Which would lead to some UML changes as well (will be explained). We decided to do our project based on the MVC (Model-View-Controller) framework. So, we separated the logic into three interconnected components, which helped in in organizing the code in a more manageable, efficient, and scalable manner.

- **Model:** Represents the data and the business logic of the game. This would include classes for game objects, players, levels, and any data-related logic.
- **View:** Represents the UI of the application. This would include all classes related to rendering the game on the screen, such as GameView, MenuView, etc.
- **Controller:** Acts as an intermediary between the model and the view. GameCtrl, MenuCtrl, and other control classes serve this purpose.

After these changes, we moved some of our files to their new folders, and removed and added new Java classes. After that, we had our base game with every Java class needed, and just separated the roles between each other, to complete them. This new framework really helped us in understanding the project, and our roles, and how each Java class works.

UML Changes:

As it was mentioned, we had underestimated the complexity of this project. Therefore, we decided to make changes to our UML, and our overall approach. Before switching to MVC, the game was organized into a set of classes with no clear separation of concerns.

- **Model:** We had to isolate the pure data and logic parts of the game into classes that don't directly handle user input or output rendering. This would mean reworking classes like Enemy, Player, Level, and Reward to remove any presentation and control logic they contain.

- **View:** The classes that are strictly involved with presenting information to the user would be part of the View component. This means that the Game, Menu, and any rendering code had to get separated into their respective view classes, like GameView and MenuView.
- **Controller:** Our first UML design showed direct interactions between user input classes (MouseInput, KeyboardInput) and the main game processing classes. After switching to MVC, classes such as GameCtrl, MenuCtrl, and InputHandler would be part of this category, processing input and updating the model or the view accordingly.

This transition would require us to:

- Refactor existing classes to fit into the correct category of MVC. For example, Game, which contained business logic, rendering, and input handling, would need to be split into GameModel, GameView, and GameCtrl.
- Introduce new classes or interfaces as needed to support the MVC architecture, such as BaseView, GameModel, or CtrlFactory.
- Modify relationships between classes to reflect their new roles. For instance, user input classes would no longer directly update game objects or render graphics; instead, they would communicate with controllers.
- Realign Methods that were previously directly handling data and user interaction to fit the MVC patterns.

Management Process:

Our management process was a blended approach of agile methodologies where roles were clearly defined by flexible. For instance, Ali focused on the model components due to his strong logical implementation abilities whereas Steven took charge of the View component. We used Discord for daily check-ins to resolve blockers and facilitate a continuous and collaborative development environment. Our Discord channel acted as the focal point for all project-related conversations, effectively communicating each team member's responsibilities whereas GitHub was used for code management. Our in-person meetings, held once a week, were crucial for brainstorming sessions and resolving complex issues that were challenging to communicate remotely.

External Libraries:

The Java Standard Edition API was used, which is included with the Java Development Kit (JDK). For instance, in the GUI, we used javax.swing, because of its extensive widget toolkit which was flexible and powerful. For example, using 'Jpanel' and 'Jbutton' were instrumental to constructing an intuitive main menu screen with minimal hassle. Also, compatibility with the MVC development pattern was also a key factor when choosing to integrate Swing components.

Quality of Code:

We did regular code reviews in our development process and occasionally developed in pair programming sessions, which supported an environment of collaborative and constructive feedback. We also employed a checklist approach during code reviews homing in on code efficiency, readability, and alignment with MVC principles. We were also locally communicative, and aware of our process and codes (mainly through discord, and GitHub). Before pushing, we approved each other's codes, and gave opinions and feedback, to achieve the best quality possible.

Challenges Faced:

Adapting to the MVC architecture was our most significant challenge. The change required a thorough rewrite of our initial plans, and a thorough comprehension of the MVC pattern and how it applies to our game.

We had trouble keeping up with the growing features of our game before moving to MVC. The connection of game logic and user interface elements resulted in our codes to be challenging to expand and maintain. It was more difficult than expected to monitor bugs and implement new functionality because of this complexly constructed structure.

Moving to MVC provided new difficulties, especially about the learning of it, and its setup. Additionally, we needed to make sure that there were no problems in the communication between the controller, view, and model components, which meant lots of communication between the team members, watching and reading instructions of MVC, and so on.