Overall Approach:

Going into the project, we had minimal experience developing a game, so we consulted a playlist from the YouTuber RyiSnow called “How to Make a 2D Game in Java” to learn the basics of what we needed to do. This tutorial by no means was the only answer we needed to figure out what to do and how to do it, but it let us start learning what and how to do it. We started with implementing the various elements that would be placed around the board like goblins, keys and the lever; this created a path for us to work with since the whole game was based on interactions the player would have with these various elements. The next step was implementing the player and how they would interact with user inputs and other objects around the board (CollisionChecker). The goblins were a particularly challenging element to implement, given that they create a new shortest pathway by tracking the player constantly. We then made a map in the text editor and added the related sprites, this took a lot of trial and error since it is harder to visualize what the map would look like in the text editor. This implementation made us rethink how the game would be implemented and led to the creation of the MapGenerator class.

Changes from phase 1:

While drawing the UML class diagram for phase 1, we tried to come up with all the different elements we wanted to incorporate into the game and made the diagram by finding a logical way for these objects to interact; this led to a blueprint that was roughly the same as the final product but we made sure to add classes and separate overly complex classes for simplicity.

The few changes we made from the UML diagram were to move punishments to the Entity class (StaticEntity in UML) because in practice, punishments behaved similarly to bonuses and rewards, where if a collision between the player and an object from Entity, there would be a deduction or addition to the score or keys collected. The biggest change was adding the MapGenerator class that handled creating and placing the maps, static and non-static entities; after initialization, it is also responsible for updating and keeping track of the score and player position, the number of keys required to open the door and opening

the door; this allowed us to combine Level, Board, Position and MazeGame classes from the UML diagram.

Power boosts were abandoned and not implemented because they did not make sense given the context of the game. The speed boost, for example, would make the game too easy and would be overpowered, and on the other hand, if the power-up were only a marginal improvement, it would be meaningless. There was no perfect balance that we could come up with where power-ups made sense to implement.

A second level where the player would enter once they pass through the exit was one of our big ideas at the beginning of the project, but once we realized how difficult implementing the first map was and how much effort and time it took to make sure all the elements acted as desired, we quickly realised that implementing multiple maps with different themes, elements and goblins would require a lot more time and could lead to our code having many more issues and more of our time would be dedicated to debugging.

The use cases described in phase 1 also saw changes, mainly the Save Game, Configure Controls, Use Super Points and Complete Level cases. Save Game never felt required because it was apparent once we realized that the game would not have multiple levels or complex progression tracking. Configure Controls was not required once we implemented User inputs; WASD or arrow keys were the only buttons needed to play the game. The Use Super Points case relates to the implementation of power-ups, and we decided to omit these from the game; the use case became non-existent; the Complete Level use case also became non-existent once a next level was omitted.

To improve efficiency in storing and retrieving game objects, we used a HashMap<String, MainObject> in ObjectManager.java. The keys are coordinates in x,y format, allowing us to access objects in constant time O(1). This data structure was chosen because it eliminates the need for linear searches and supports fast additions and removals, which is crucial for performance.

The overall idea of the game ended up staying close to the original idea: goblins would still chase the player, collect keys and rewards, flip the lever and exit through the door to win the game. The major changes we made were omitting the idea of power boosts (for example, double speed), the possibility of multiple levels, and the MapGenerator class.

Management Process

We created a Discord group chat and arranged meetings twice a week to communicate and catch up with each other. We initially divided all the work into 5 subcategories as follows:

- Environment Setup
- Movement Systems and AI Systems
- Visual Systems
- Collision Systems
- Game States & UI

However, we quickly realised that this method was not practical, and once we had the basic building blocks of the game, we just started to ask each other what needed to be done and helped each other out. This corresponded with our updated responsibilities; once we started to collaborate and help write code in each other's sections, our cohesion and understanding of the code and each other improved. We might not have been the best organized in terms of splitting our responsibilities and working on each other's code individually at the start, but we learnt to work with each other and share responsibilities, which made our code and our group more functional.

External Libraries

We did not find any need to use external libraries to make our project operate. We found that the internal libraries in Java were enough to let us implement all that we wanted to.

Code Quality

We did not organize our code as we wrote it because all our code and structure was not stable and always operated like we needed to make huge changes. Aiming towards organizing the code when we were not even sure of its usability and reliability felt pointless. We, as a group, always coded to implement the functions and make sure they work as desired; while going through this process, we made sure to add comments and descriptors that were concise and clear enough so that our teammates understood what the function of a particular block of code was. Concerning software engineering principles, we believe that we could have done better at accurately encapsulating functions and classes; as we focused on implementing and making sure functions worked as desired, we left encapsulating to a later organizing stage where we considered functions' requirements and the appropriate encapsulation for it. Some attributes that could be private were made public because, in different iterations of the code, they might have needed to be accessed from other classes.

For example, to improve movement constraints and ensure proper interaction between game elements, we implemented a centralized CollisionCheck class. The system detects three types of collisions: tile-based collision, preventing movement into walls, entity collisions, ensuring enemies do not overlap and detecting player-enemy interactions, and object collisions, allowing the player to collect items and activate switches. By handling all collision logic in one class, we maintain clear code and easier debugging.

Biggest Challenges

When we first started working together, we severely struggled at communicating. We made a Discord group and set up meeting times at the start, but everyone ended up having heavy schedules that conflicted with the meeting time slot. We got better at keeping each other accountable and responsible by creating a new list of responsibilities halfway through the phase and reasssigning responsibilities. This second list and conversation about accountability had a huge impact and made our group more cohesive and focused.

Merging code was a challenge we faced towards the end of the phase since we all ended up coding on our own branches and helped each other out. Our code tended to overlap sometimes and cause conflicts, these conflicts made us change our approach to certain issues like the pathfinding algorithm. We later made sure to consult each other when working on each other's code to make sure we fixed these conflicts and chose which approach was better at solving our problem. Although this issue was reduced with our collaboration and teamwork, sometimes conflicts tended to happen that required us to debug code more often.

Goblin AI was challenging to implement. We implemented a modified A pathfinding algorithm* that dynamically calculates the shortest path to the player. Each tile in the world is represented as a node, with values for G-cost(distance from start), H-cost (heuristic to goal), and F-cost (total cost). The goblin selects the tile with the lowest F-cost at each step, ensuring optimal movement.

To handle obstacles, if a collision is detected while following the path, the goblin adjusts direction to find an alternative route. This system allows goblins to effectively navigate the game world while avoiding walls and other static objects.

Similarly, CollisionChecker gave us a lot of problems from the beginning. We observed with our first implementation of CollisionChecker that even when the player and goblin were on the same tile, a collision would not be detected. Brainstorming together, we decided to go back to basics to fix the bugs. We used print statements to verify bounding box calculations, then we moved to drawing the collision boxes separately on top of the players and goblins to observe what was happening. Fixing this algorithm required all of our collaboration and was the source of most of our bugs.