Group #13 - Phase 2 Report

Xuyun Ding, Denzel Tjokroardi, Kyle Tsia, Yihao Wang

Implementation & Logistics:

Our overall plan of attack for this project was to follow an adjusted Spiral Model, such that it was adjusted because we do not actually have a client, so the feedback stage was mainly just done within our group. Now that we are in Phase 2, we have passed the "Communication" and "Planning" phases of the model, as we were now on the same page in terms of functionality, requirements, and goals, and we were heavily in the "Modelling" and "Construction" phases. Each "Deployment" phase would create a new iteration of the model that was an increase in product size and complexity.

We began this phase by dividing each individual class that we planned on implementing and agreed to complete our individual components by a set date. After review, this method proved to be faster but yielded less quality code and increased work when it came to aggregating the parts together. Alternatively, we scheduled time to meet as a group and worked through the program together as a team. The latter method allowed for more readable code and a more effective development process, despite being more time-consuming. However in light of the COVID-19 orders of social distancing, we progressed to pairs and trio coding over online communication, this became another challenge to overcome, as ineffective communication and conflicting schedules were a major setback.

We primarily used the AWT package as it included capabilities for painting graphics and images neatly. It allowed for ease of changes in rendering throughout the game running threads, all while holding onto constants, like the game board image.

Division of Roles & Responsibilities:

Denzel Tjokroardi: Implementation of enemy class and scoring system for enemy class along with the GUI end-screen and lose condition for moving enemies.

Xuyun Ding: Construction of the maze, collision detection between the player and the wall, icon images for different entities, starting/instruction screen and restart function.

Kyle Tsia: Implementation of reward and bonus reward objects, implementation of win condition.

Yihao Wang: Initialization of the game board and its GUI, initialization of the player and its movement, and the implementation of moving enemies.

Code Style & Enhancements

In order to make our code as understandable and concise as possible, we followed Java coding conventions. Moreover, we added short comments where necessary to make our code easier to read and trace for reviewing and debugging later. As for now, we are aiming to improve and optimize the structure of our code to possibly simplify the logic.

External & Internal Libraries

In this phase of the code implementation, our group didn't import any external libraries for compacting game functions. Most of the aspects in our game were realized by just importing the internal libraries already provided by Java language, such as the *java.util* package, *java.io* package, *java.awt* package and *javax.swing* package. For more detailed information, please see below in the Modification & Extension part, which shows how we handled some essential elements of the game including the GUI, maze construction, collision detection, reward spawn and the movement of the enemy and the player.

Modifications & Extensions:

As a result of testing and discussing foreseeable problems and difficulties, much of our game changed from the original planning phase. However, our base concept remained more or less the same.

Game Board and Appearance

Originally, our plan was to create an arraylist of arraylists in which we map each element within this matrix to an image depending on the value it holds. For the sake of simplicity, we decided to use a preprocessed image of the game board and use the awt package to interpret and render each cell. This change extends to our original design of each cell. In our phase 1 planning and modelling, we were intending to make each individual cell hold a value, such as *Enemy, Reward,* or *Barrier*, however, after our change in the gameboard and the need for more complex extensions of each, we decided to remove that idea entirely. Instead, we created a separate class for each entity on the board, and they would be rendered onto the gameboard separately and independently.

In the original concept, we were going to print the game time and game score on the board, however with a change in how we wanted the user to play the game (see "Rewards and Win-Condition"), we opted to only display game score and added a game round print. Later iterations of the game could reintroduce the game time display.

Rewards and Win-Condition

Instead of how we had modelled it in our class diagram, *BonusRewards* and *MandatoryRewards* were no longer subclass extensions of a rewards superclass. In our final design, we still extend *BonusReward*s as a subclass, however it was a subclass to the superclass of mandatory rewards. Both of these now utilize random generation, unlike how we planned it where only *BonusRewards* would have been randomly generated. Finally our fundamental win case is slightly different.

Originally the plan was to have the user collect on mandatory rewards, and they would need to find the exit cell to lead to a new map. However in the sake of simplicity, we decided that upon collection of all the mandatory rewards, the user would simply spawn a new set of enemies and rewards, and they would win the game after a set amount of rounds completed. This was implemented for simplicity, and for the possibility of extending the game to have a "speedrunning" component, whereas the user could race to see how many rounds they can complete in a set amount of time.

Player
For this particular part, we initially used the single-color pixel to represent the player, and will replace it with a more visualized picture later to match our UI mockups in the design phase. We decided to create four boolean variables to indicate and allow the movement of the four directions and combine with the *tick* method to work with the movement speed of the player. Also, we added another boolean method to detect whether the movement of the player will collide with the wall of barriers based on the idea of class *Rectangle* in Java. Additionally, we chose to keep track of the player's current score here for *Game* class to get the value and show on screen.

Challenges & Problems:

Visualization
One of the main challenges we needed to overcome was to choose a proper way to visualize the different objects in the game. We decided to use various colors of pixels/rectangles to differentiate one from another (*Walls, Barriers, Enemies, Rewards,* etc.). Moreover, in later iterations, we plan to add some pictures and avatars to represent those different elements in the game to make it more user friendly and appealing. However with the addition of this non-functional requirement, it introduces a new element of difficulty, namely in collision detection (see below).

Collision Detection
Collision detection between the player to the board was simple. However collision detection between the enemies and moving enemies to the player was difficult. Checking for touching had already been hard when we were just checking on linear and simple-shaped objects, however in later plans to implement non-linear and complex-shaped objects, this increases the level of difficulty. We will need to detect the collision on pixels of the object that may vary from object to object.

Project Synchronicity
Throughout the project, especially after the COVID-19 announcement of social distancing, it became difficult to work on the project in separate parts at the same time. Building different components of the game while others were still adjusting the underlying structures or attempting to merge onto a changed master code proved to be difficult and frustrating at times. We overcame this by pushing often into our branches and submitting merge requests into the master

branch as soon as we had a block of code that worked. This way, we could be as efficient as possible because our team members could have updated code as quickly as needed.

Enhancements

Current

Within the current game code, we made active decisions in order to allow for cleaner, more readable code. The first being direct comments in the code itself, this allowed us to document our process not just for ourselves but for our team members looking at our parts as well. The second was utilizing Javadoc documentation on all of our methods in all of our classes. This again allowed for us to keep track of our process for both ourselves and our teammates.

Future

We have a few ideas to enhance our game in the future. The first, as stated in the "Modifications & Extensions" section, could possibly extend the game to include a speedrunning setting. The user would play as many rounds as possible within a set in-game timer. Off of this, we could add a high-score feature that records past run times. A second extension could be customizable images for each of the in game objects. For us this would just be as simple as having a folder of pre-built images and changing the path mapping the object image to whichever the user selects. A final extension could be allowing for a change in difficulty. An increase in difficulty could mean increasing the amount of enemies on board, increasing speed of the moving enemies, increasing the size of the board, or making the rewards move. Realistically, after we created this base game, the extensions off of it are endless.