Phase 2 Report

Group 25

Vincente Buenaventura
Noah Kamzelski
Samin Moradkhan
Lionel Song

Approach

- Looking at UML diagram
- Researching about libraries to use (Using JFrame)
- Coding parent classes first, then children classes
- Figuring out how to code something so that it shows up in the window
- Leaving comments in code for other teammates to understand
- Finding out what to use for sprites in our game
- Which IDE should we use
- Splitting up tasks so people know what to start on first
- Communicating with pushes, bugs, needing help, project requirements, etc.
- Reading a lot of documentation

As a team, we approached Phase 2 first by going through steps and figuring out a timeline and map of what should be done. We needed to figure out a lot of things before even coding which is why we resorted to group meetings to get those things figured out. After a few group meetings, we completed the requirements that were needed to start coding the content of the game. Once roles were assigned and tasks were given to each person, no more group meetings were needed because communication via text message was sufficient. Lots of research was essential in making the game because not many of us were familiar with the libraries in Java. Apart from the concepts, material and syntax we learn in class, there wasn't enough information to start coding the necessities of our game. We had our phase 1 to look at and guide us at the start but we needed more knowledge around making games in Java. Once we all figured out what general components were needed we were able to communicate those needs with each other and start coding from there. Aside from coding the main general things needed for a game to work in java, we tried our best to implement the concepts learnt in class.

Adjustments and modifications to the initial design

We have added sound to our initial design so when the player collects the rewards or falls into traps there is a sound effect for it. We have two sets of keys to control the character movement which are the arrow keys and the WASD keys on the keyboard. When the character has fallen into a trap it will lose score as well as starting from the start position all over again. We initially planned to only deduct from its score but we thought it would be better to have the player start again. In our design right now the player is able to win when it reaches the end point even if it has not collected all the rewards which can be changed later to only let the player win when it has collected all rewards and then reaches the end point. Everything else follows our design suggested in phase 1 with some enhancements to the UI.

Libraries and why we used them

```
javax.imageio.ImageIO;→ reading an image
java.util.Random; → implementing random for random placement of objects
java.util.List;→used to create lists (for our objects)
java.util.ArrayList; → list of enemy
javax.swing.*;→ lightweight Java graphical user interface (GUI) widget
toolkit that includes a rich set of widget
javax.sound.*; → used to import and play sounds during playing the game
java.awt.Color; → to be able to choose colour for the text and background
java.awt.Font;→ importing different fonts for the UI
java.awt.Graphics2D;→ using graphics to draw to the screen
java.awt.image.BufferedImage;→ used to load and read images to the screen
java.awt.event.KeyEvent → keyboard interaction
java.awt.event.KeyListener; → keyboard interaction
java.text.DecimalFormat;→ to be able to show decimal format for time
java.io.BufferedReader;→ Reads text from a character-input stream, buffering
characters so as to provide for the efficient reading of characters, arrays, and
lines
java.io.InputStream;→ used for reading and it could be a file, image, audio
java.io.InputStreamReader;→ is a bridge from byte streams to character
streams: It reads bytes and decodes them into characters using a specified
charset
java.io.IOException;→ to catch exceptions when we use "try"
```

Management Process and roles

Our general approach to this phase was to divide roles based on the phase 1 UML diagram. We had our first meeting to divide tasks and work on our preferred parts of the project. Then during the implementation process we also had a second meeting to show our updates and explain the part we worked on. Our main communication format was in discord where we mentioned which part we were currently working on, if we pushed to the repository, found a new bug or fixed a bug, or any design question was shared in the chat. We communicated pretty well and everyone was on the same page while implementing the code. However below is a more detailed breakdown of our tasks and what each person mostly worked on.

- Lionel
    - Created the Animate and Inanimate entities Classes which extends the Entity class also created by me.
    - Created the Coordinate Class
    - Created on the player class as well as the enemy
    - Created the obj_apple class
    - Worked on the placement of rewards on the map
    - Created the checkCollision Class
    - Updated the map to look like a laboratory
    - Created object sprites for player and enemy in the resources folder
    - General debugging
    - Writing the report

- Samin
    - Created the keyboard class
    - Created the main class
    - Collaborated on the Simulator Class
    - Created the sound class and added the wav files needed to the sound folder
    - Worked on the Screen and display (Game over and retry options)
    - Created some object sprites in the resources folder
    - Created the Tiles class and worked on the tiles_controller
    - Worked on adding the object banana to the map randomly
    - General debugging
    - Writing the report

- Vincente
    - Worked on the checkCollision class

    - Worked on the simulator class
    - Added the player score to the screen
    - Added the sound for losing the game to the sound class
    - Created the obj_trap class for the traps
    - Created the object sprite for the traps
    - General debugging
    - Writing the report

- Noah
    - Worked on the simulator class
    - Created game states
    - Created the UI class
    - Created the obj_heart class which shows the player's life on the screen
    - Added the pause state
    - Worked on game over state
    - Added timer to the screen
    - General debugging
    - Writing the report

Enhancement of the code quality

The final implementation of our game differed moderately compared to our initial design of the UML. Below explains some key changes that were different from our UML diagram to enhance code quality.

While implementing the base of our game, we realised that a simulator that updates entities and UI based on a while loop would be inefficient and problematic. A while loop meant the concept of time would be difficult to implement, since any lengthy processes (such as processes that waits on user input) would "freeze" the game. Instead, we created a Thread which simulates time by updating processes of the game independently without interruption.

Another modification we made was the separation of updating entities and updating UI. Initially, we wanted our simulator class to update entity data and UI changes in parallel. For example, when the player moves, the simulator would call the respective entity to handle both the data and UI update. The shortcoming of this idea is that updating data and UI uses different external libraries of Java. This meant that combining both data and UI updates increased coupling and decreased modularization. Instead, we opted to split data and UI data into separate functions. The simulator game loop would first update all entities before drawing them on the

UI. The increased modularity ultimately made debugging and maintainability much easier.

Lastly, we realised that multiple classes in the UML failed to follow the principles of modularity, and could be split into separate classes. One notable example is the Player class, which used to handle all collisions between entities and had an internal keyboard listener. To decrease coupling between entity classes and increase modularization between the game simulator, the Player class was made to include separate Keyboard and CheckCollision classes. These classes were located in separated modules and had significantly less coupling and cross-module interaction.

Challenges

- Not finding any good sprites online, thus having to make our own
- Bug fixes (Expand on this, talk about specific bugs)
- Some git issues (merge conflicts)
- Finding audio file to match our game theme

Making the rewards to spawn randomly on the map without being placed on top of each other or in locations we did not want:

One notable challenge we faced during the implementation phase was preventing rewards and traps from spawning on unwanted tiles. In our game, rewards should not spawn on the player and collidable tiles (such as walls and traps). Traps should not spawn on the player and bridges. The map contained three separate classes; tiles, objects, and entities. Each of these three classes had its corresponding subclasses. The problem was that there was no class which handled the interaction between all three super classes. Therefore, we had to create a new CollisionChecker and EntityList class to overcome the challenge. The CollisionChecker class handled all collisions between the three super classes. It was responsible for iterating through the EntityList class, which contained a list of objects and entities, and then checking if any of them collided with an undesirable tile. Ironically, this feature which we initially thought would be easily implemented became one of the biggest challenges we had to overcome.