Overall Approach

We began implementing the game by creating a *JPanel* class and a *JFrame* class to ensure that we have a board to draw our game with the PaintComponent() method and a window to display it. Then, we set up different states of the game and added functional buttons so that by pressing a button, it would lead the user to another state, such as instruction screen, starting a game, and exiting a game. Next, we added a *Player* class and made sure *Player's* movement is controlled by the arrow keys. Once *Player* was functioning properly on the game board, we coded a 2D array that stores the values of the maze map in the *Cell* class, where 0 indicates a wall and 1 indicates a valid cell. After the map was set up, we added wall detection methods so that the Player cannot move into a wall cell.

With a functional map, we were able to start implementing the rewards and punishments. We created an abstract *Item* class that has multiple abstract methods to be shared by *Reward*, *Punishment*, and *Bonus* classes and it is also a superclass of these three. Inside the *Cell* class, we created an ArrayList for each of these three classes and used the ArrayList methods to either add them to the game board or remove them from the game board. Once the rewards the punishments were working, we added score, a winning condition, and a losing condition and if either one was triggered, it would lead the user to its respective state. Lastly, we implemented the *Moving Enemy* class and added its tracking method that calculates the shortest path to catch up to *Player*. Overall, we successfully executed all the required functions for the game.

UML Modifications

1. Initially in phase 1, we had a Character superclass, extended by both Player and Enemy classes (Figure 1). However, in phase 2, we removed the Character superclass and created independent Player and Enemy classes that have their own fields and methods instead of inheriting from a superclass (Figure 2). This change was made because the Player and Enemy behave differently from each other. Even if we decide to make them subclasses of a Character class again, it would be easier to do it now since now we know what these two classes have in common.

2. Another change that we made is that initially we wanted to implement all the rewards and punishments inside the Board class (Figure 1). Yet, in phase 2, we created separate classes for regular reward, punishment, and bonus reward because according to separation of concerns, we should divide our software into addressable modules. Therefore, instead of having everything inside one class, we took out those three components and divided into three parts, or in this case, three classes (figure 2).

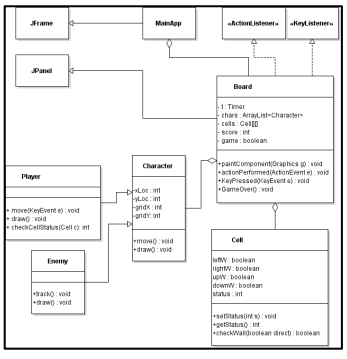

Figure 1. Phase 1 UML

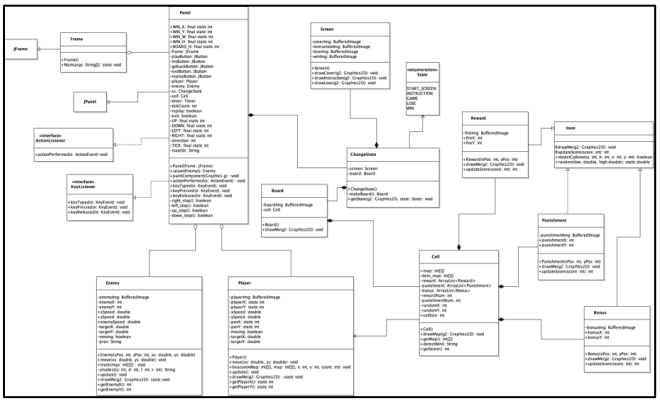

Figure 2. Phase 2 UML

Use Case Modifications

We ended up not having enough time to do everything we originally planned and some use cases have been modified. There are no doors and keys so the exception for the player to be able to go through walls with doors no longer exists. Additionally our bonus reward no longer has the potential of being a trap and is therefore always a positive reward. All other use cases stayed the same.

Roles and Responsibilities

We had meetings twice a week during phase 2 to ensure everyone was on the right track and work was produced according to our plan. We all worked in synchrony and built things in a step by step manner. We did this by creating expected prototypes which are as follows:

Prototype 1:
- Button/Title screen functioning
- Implement Cell
- Generate pixel board with just normal cells – with an entrance and an exit
- have player at entrance and enemy at exit

Prototype 2:
- Get player and enemy moving properly
- Generate inner Walls/Maze
- Generate points

Prototype 3:
- Create shortest path method for enemy (extend this to the board builder)
- Clean up loose ends, have displays showing properly.

Brian – Completed the full implementation for the Enemy's movement method and helped devise strategy for implementation. Improved code quality for generating rewards and punishments.
Jasim – Coded the game timer, scheduled group meetings, and assisted in game design logic.
Kevin – Assisted in coding the timer, the enemy, player classes. Assisted in writing the report.
Vera – Was the artist and brains of the project. Spearheaded the report and code production, including wall detection, player, screen, state, cell, reward, punishment, bonus, and item classes.

External Libraries

We did not use any external libraries in our code. However, we used a number of Java core libraries to complete the project. Java core libraries that we used are:

1. **java.awt.event**; the ActionListener methods this library provides allow the user to interact with our system by clicking different buttons, such as Play, Instruction, and Exit with a mouse. The KeyListener methods allow the user to control the player with arrow keys.
2. **java.awt.Graphics2D**; this library allows us to display images and draw shapes.
3. **java.awt.image**; this library allows us to load images into the game so that we can display the characters, rewards, and game backgrounds with different images.

4. **java.util.ArrayList**; this library allows us to create different ArrayLists for storing regular rewards, bonus rewards, and punishments and to control their behaviour with ArrayList methods such as add() and remove().
5. **java.io/java.imageio**; this library allows us to catch any input or output exception.
6. **javax.swing**; this library allows us to add JButtons to our game, draw graphics on JPanel, and display everything in a JFrame window.

Improvement

1. Initially, multiple constants were created to indicate different states of the game, including START_SCREEN, INSTRUCTION, GAME, WIN, and LOSE (Figure 3). We realized this is repetitive, not as maintainable, and can often get lost in a pile of code. Therefore, to improve the quality of our code, we created a new enumeration class (Figure 4) to hold these constants. By singling out these constant variables, we emphasized the importance that these variables have in our game since it is essential to have different states in a game. Along with this change, we also modified the if-else statements (Figure 5) that would switch the state of the game into switch cases (Figure 6) that use the State enumeration class to do the job.

```
private final static int START_SCREEN = -1;
private final static int INSTRUCTION = 0;
private final static int GAME = 1;
private final static int WIN = 2;
private final static int LOSE = 3;
```

Figure 3. Multiple constant variables

```
public enum State {

    START_SCREEN, INSTRUCTION, GAME, WIN, LOSE;

}
```

Figure 4. State enumeration class

```
public void getState(Graphics2D g2) {

    if(state == START_SCREEN) {
        System.out.println("In state START_SCREEN");

    }else if(state == INSTRUCTION) {
        System.out.println("In state INSTRUCTION");

    }else if(state == GAME) {
        System.out.println("In state GAME");

    }else if(state == LOSE) {
        System.out.println("In state LOSE");

    }else if(state == WIN) {
        System.out.println("In state WIN");
    }
}
```

Figure 5. if-else statements

```
public void getState(Graphics2D g2) {

    switch(state) {

        case START_SCREEN:
            System.out.println("In state START_SCREEN");
            break;

        case INSTRUCTION:
            System.out.println("In state INSTRUCTION");
            break;

        case GAME:
            System.out.println("In state GAME");
            break;

        case LOSE:
            System.out.println("In state LOSE");
            break;

        case WIN:
            System.out.println("In state WIN");
            break;
}
```

Figure 6. switch cases

2. Second improvement that we made to enhance the quality of our code is combining regular rewards and punishments into one ArrayList rather than two separate ArrayLists (Figure 7). This change made our code more reusable and less redundant because we are able to control both rewards and punishments using only one ArrayList (Figure 8).

```
for (int i = 0; i < reward.size(); i++){
    Reward rewardi = reward.get(i);
    rewardi.drawMe(g2);
}

for (int i = 0; i < punishment.size(); i++){
    punishment.get(i).drawMe(g2);
}
```

Figure 7. Two separate ArrayLists

```
//controls all items
for (int i =0; i < items.size(); i++) {
    Item itemi = items.get(i);

    //display each item
    itemi.drawMe(g2);
    if(itemi.detectCollision(Player.getPlayerX(), Player.getPlayerY(), itemi.getPosX(), itemi.getPosY())) {
    score = itemi.changeScore(score);
    // remove the item that Player touches
    if (itemi instanceof Reward) {
        collected++;
    }
    items.remove(itemi);
    // reset the value of the cell back to 1
    item_map[Player.getPlayerX()][Player.getPlayerY()] = 1;
    }
}
```

Figure 8. One ArrayList

3. Another improvement that we made is that originally we had several drawMe() methods inside the *Panel* class for drawing screen backgrounds for different states of the game. However, we realized that the *Panel* class has other fields and methods that are irrelevant to the screen backgrounds. As a result, following the principle of separation of concerns, we decided move out the drawMe() methods and store them in a newly created *Screen* class (Figure 9), where the class is a module that contains solely these drawMe() methods for drawing different screen backgrounds.

```
public void drawCover(Graphics2D g2){
    g2.drawImage(coverImg, 0, 0, 600, 700, null);
}

public void drawInstruction(Graphics2D g2) {
    g2.drawImage(instructionImg, 0, 0, 600, 700, null);
}

public void drawLose(Graphics2D g2) {
    g2.drawImage(loseImg, 0, 0, 600, 700, null);
}

public void drawWin(Graphics2D g2) {
    g2.drawImage(winImg, 0, 0, 600, 700, null);
}
```

Figure 9. drawMe() methods

Challenges

A large challenge was getting used to and using the core libraries since for half the group this is the first time they have coded with java. Developing the movement method for the enemy took a bit of time to draft out and implement due to the logic. Another large challenge was sticking to the original UML. This was likely because we had naively drafted things, for example, just because the enemy and the player are characters does not mean they should both extend the same superclass. I believe all these issues boil down to a lack of experience in developing a program of this size; however, to overcome this challenge, we studied relevant materials outside of class and watched YouTube videos and in the end, we successfully produced a playable maze game.

Another challenge that we faced is time restrictions. A lot of the additional ideas from phase 1 were never implemented because we did not have enough time. For instance, we only have one map at the moment and are not randomly generating new maps every game. We also only have one level instead of several levels where the game difficulty increases in each level. We would also like to use more design patterns to enhance the quality of our code if we had more time implementing the game.