Dylan Young, Sajandeep Toor

HW3 Report

One of the biggest problems we noticed was the GameController class, it suffered from **being a class too large and doing too much**. The problem with this class was managing the state of the game, generating all the entities (traps, rewards, enemies, etc), setters and getters for all the entities, managing the gameplay related threads, handling game input and logic. This class was simply doing too much and had over 1,000 lines of code. It was also difficult to navigate and find the methods you were looking for. This was a major flaw in our design.

The solution to this was splitting the class up into multiple other classes. We split GameController into Entities, EntitiesGenerator, GameController, GameInput, GameLogic and Timer.

- Entities: managed the getting and setting of entities and it could only be get from publicly and setters were protected
- EntitiesGenerator: Handled the random generator of entities
- GameController: Managed the state of the game, example pause state and threads
- GameInput: Handled the player's input
- GameLogic: Handled collisions, expiry of bonus rewards, hasWon and other logic methods
- Timer: The game timer to keep track of seconds since the game started

fe03a3b0

Also, at a later point we decided to move the Maze class to its own package. This improved our **file structure**. This allowed us to use protected methods in the game package allowing information to be communicated within the game package while having these methods not available publicly, since they had no reason to be exposed publicly. af97ebd9

Another change we made was **adding an enemy class**, before enemies were the CharacterModel class, instead it made sense to move the enemy movement logic to the enemy class itself, previously it was in game controller. It made more sense the GameLogic since that was more general and responded to the player's input. It also made it more clear in the code that it was an Enemy. Enemy still extended the CharacterModel. e9e5a944

We had some **spaghetti code that was difficult to read and understand**. If another programmer looked at it, they would have a hard time trying to figure out what is going on. This is a major issue. There were some long if statements and places where a local variable to data read multiple times would've been helpful. An example of this would be in GamePlayScreen in the revalidateMaze method. Comparing Figure 1 to Figure 2, there's a massive difference in readability. Local variables for maze, cell and entities were created eliminating the long if statements. Position was used instead of x and y coordinates **eliminating otherwise dead code** and improved readability. Instead of getting the cellType itself, just call methods like isEmpty, isEnd, etc. f3da8d20

At some places, we had a **lack of documentation** where Javadocs for classes or methods were missing and some Javadocs were broken, had something missing or had spelling mistakes: a4a6b7bc, eda0061e, e6f21716

Code was also made more **concise and consistent** in eda0061e where if an if statement only had one line after the block brackets were removed, unnecessary if statements for returning were removed.

Long **variable and method names were renamed**, for example in Player class as it was a singleton there was an instance variable but it was called "playerInstance" this was unnecessary, as it was already in the Player class, instead renamed to "instance". B1ce0333

One method name that was confusing was isPath in the Maze class. Since there were already methods such as isWall, isTrap, etc. in the Maze class which looked at the CellType and returned a

boolean it would make sense that isPath would do the same for the Path cell type, nope! Instead isPath checked if there was a path between 2 points in the Maze. isEmpty handled the expected behaviour for isPath instead. Our solution was renaming isPath to isRoute and isEmpty to isPath. e671a262, feaaa7d7.

We found that some of the UI components were difficult to manipulate and many of the classes that form the UI package had long lists of private UI components. The GameUI class had JPanels variables that were never used at the same time. These **classes were difficult to maintain and difficult to read for other programmers**. We fixed this by using inheritance and making each screen their own object that extended the JPanel class. This allowed us to use 2 JPanel components for the GameUI class, one for the main screen and one for the sub screen, instead of the many JPanel instances for each screen. This allowed us to easily maintain the components on the UI and made the code more readable. bd833d48, 2a3b699f

In the SpriteIcons class, all the methods to retrieve an image had similar if statements. These if **statements were long and extremely repetitive**. We fixed this by making a method that returns a boolean of the conditions that were in the if statements for the image retrieving methods. This method takes 3 parameters that are used to form part of the return statement for the method. Also the methods that retrieve arrays of images had multiple lines of code that retrieved images of the same width and height but for different images that had different file locations. We fixed this by creating arrays of Strings in each method that contained the file locations of all the images and ran a loop to iterate through the array of Strings to retrieve the images from these locations. a5211017

Another thing that we found was **inconsistencies with formatting**. We agreed to follow the Java Code Standard for formatting (besides switch statements) and at points there were missing spaces between brackets, missing new lines between methods, missing spaces at the start of comments and long lines not broken up if possible. All these things were fixed using an auto formatter which formats the code to the Java Code Standard that we were following. 47625227

Figure 1

```
private void revalidateMaze() {
    for (int i = 0; i < mazeHeight; i++) {
        for (int j = 0; j < mazeWidth; j++) {
            if (GameController.getInstance().getPlayer().getPosition().getX() == j && GameController.getInstance().getPlayer().getPosition().getY() == i) {
                cellLabels[j][i].setIcon(playerIcons[3]);
            } else if (GameController.getInstance().containsEnemy(j, i)) {
                cellLabels[j][i].setIcon(enemyIcons[1]);
            } else if (GameController.getInstance().containsReward(j, i)) {
                Reward reward = GameController.getInstance().getReward(j, i);
                if (reward instanceof BonusReward) {
                    cellLabels[j][i].setIcon(getBonusReward(cellWidth, cellHeight));
                } else {
                    cellLabels[j][i].setIcon(getReward(cellWidth, cellHeight));
                }
            } else if (GameController.getInstance().containsTrap(j, i)) {
                Trap trap = GameController.getInstance().getTrap(j, i);
                if (trap.getTrapType() == TrapType.BOOBYTRAP) {
                    cellLabels[j][i].setIcon(getBoobyTrap(cellWidth, cellHeight));
                } else {
                    cellLabels[j][i].setIcon(getTrapFall(cellWidth, cellHeight));
                }
            } else if (Maze.getInstance().getCell(j, i).getCellType() == CellType.PATH || Maze.getInstance().getCell(j, i).getCellType() == CellType.END || Maze.getInstance().getCell(j, i).getCellType()
            == CellType.START) {
                cellLabels[j][i].setIcon(getPath(cellWidth, cellHeight));
            } else {
                cellLabels[j][i].setIcon(getWall(cellWidth, cellHeight));
            }
        }
    }
}
```

Figure 2

```
private void revalidateMaze() {
    Maze maze = Maze.getInstance();
    Entities entities = Entities.getInstance();

    for (int i = 0; i < mazeHeight; i++) {
        for (int j = 0; j < mazeWidth; j++) {
            utilities.Position pos = new utilities.Position(j, i);
            Cell cell = maze.getCell(pos);

            if (Player.getInstance().getPosition().equals(pos)) {
                cellLabels[j][i].setIcon(playerIcons[3]);
                continue;
            }

            if (entities.containsEnemy(pos)) {
                cellLabels[j][i].setIcon(enemyIcons[1]);
                continue;
            }

            if (entities.containsReward(pos)) {
                Reward reward = entities.getReward(pos);

                if (reward instanceof BonusReward)
                    cellLabels[j][i].setIcon(getBonusReward(cellWidth, cellHeight));
                else
                    cellLabels[j][i].setIcon(getReward(cellWidth, cellHeight));
                continue;
            }

            if (entities.containsTrap(pos)) {
                Trap trap = entities.getTrap(pos);

                if (trap.isBoobyTrap())
                    cellLabels[j][i].setIcon(getBoobyTrap(cellWidth, cellHeight));
                else
                    cellLabels[j][i].setIcon(getTrapFall(cellWidth, cellHeight));
                continue;
            }

            if (cell.isEmpty() || cell.isEnd() || cell.isStart()) {
                cellLabels[j][i].setIcon(getPath(cellWidth, cellHeight));
                continue;
            }

            cellLabels[j][i].setIcon(getWall(cellWidth, cellHeight));
        }
    }
}
```