Final Project Step - 6, Reflection

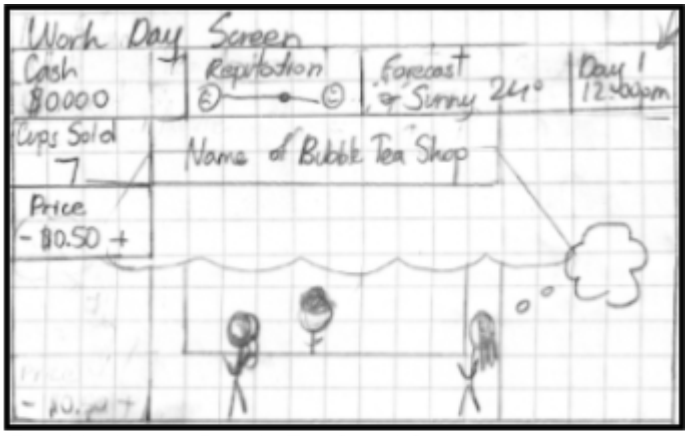
Sarah Simionescu

Introduction

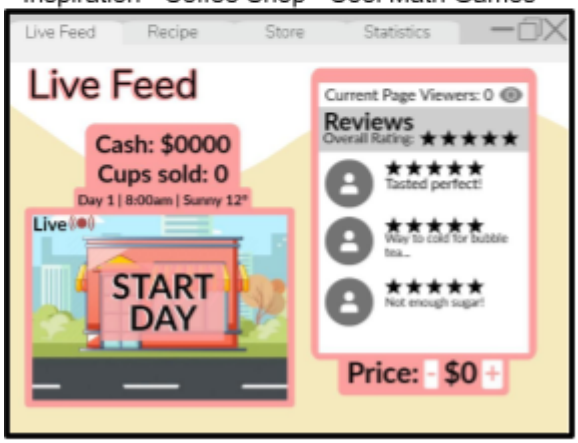
Developing and creating this game came with countless challenges that have allowed me to develop my skills as a programmer and deepen my understanding of Java. In this reflection, I will highlight three major challenges that have completely changed my approach to accomplishing this project.

Game Concept and Design

Inspiration - Coffee Shop - Cool Math Games



First Mock-up in Bubble Tea Stand Style



Second Mock-up in Online Shop Style

Final Game

In my original mock-up, I planned for the bubble tea shop to be run as a stand. Customers would physically walk past the stand and comment using speech bubbles. In my second mock-up, I designed an online shop version of the game.

Bubble Tea Stand Style	Online Shop Style

- It was too graphically challenging for my skill set and time constraint. - It looked hectic and confusing. - It wasn't very original.	- I aesthetically separated the information into several tabs. - Delivery cars and comments were much more realistic to animate.

Separating Graphics and Game Functions

When creating my Card Game, I learned how to update graphics and game operations simultaneously in individual Threads (Game Class and Display Class). There was just one problem: I could only respond to user input in the Display class since mouse and key listeners use the same JFrame as the graphics component.

```
public DisplayScreen(Display d) {
    D = d;
    G.gameMode = 0;
    newGame = true;
    run = false;
    aCounter = 0;
    windowManager();
    addKeyListener(keyboard);
    addMouseMotionListener(mouse);
    addMouseListener(mouse);
    addMouseWheelListener(mouse);
    G.start();
}
```

Display window and listeners for user input all created within the same JFrame object.

Therefore, if I run game functions directly from my Mouse class, the graphics will stop updating (frozen screen) until the game functions are complete. To solve this issue in my Card Game, my Game Class would simply wait and check the Mouse Class every couple of milliseconds to see if the mouse was pressed recently.

Card Game - Mouse Class to Game Class Communication

Mouse Class setting mouseClicked to true when the mouse has been clicked in the past 70 milliseconds.

```
public void mouseReleased(MouseEvent e) {
    if (D.run == true) {
        mouseClicked = true;
    }
    try {Thread.sleep(70);} catch (InterruptedException z) {}
    mouseClicked = false;
}
```

Game Class waiting until mouseClicked is true to respond to the user's input.

```
if (g == 0) {
    while (D.mouse.mouseClicked == false || checkMenu() == null) {
        pause(50);
    }

    if (checkMenu() == 0) { //main menu
        sortMethodMenu();
    } else if (checkMenu() == 1) {
        sortTestMenu();
    }
```

This method, of course, is quite finicky. After running many tests in a separate project, I came up with a solution. The Mouse Class initiates the Game Class Thread independently, directly when it receives user input, by reinitializing the class and initiating the Thread's run() function. No more waiting and checking, the Game Class can react to the user's input without pausing the graphics component.

Bubble Tea Game - Mouse Class to Game Class Communication

```
public void runGameMode(int i) {
    D.G.gameMode = i;
    game = new Thread(G);
    game.start();
}
```

Mouse Class initiating Game Thread directly and independently from the Display Class.

Now that I had better communication between my Display and Game class, I clearly outlined the responsibilities of each class. Any other class is just a tool or an "extension" used by these classes.

Two Main Parallel Threads	
Game Class	**Display Class**
- Stores long term variables - Updates variables based on user input - Initiates UI Components	- Stores short term variables for animations and displaying statistics - Continuously updates variables through paint() function - Reacts to user input to initiate Game Thread

Taking Advantage of UI Component Interface

Since this game required many buttons, text and graphics, I decided to organize these components and create universal methods for simplicity and efficiency. I developed the UI Component interface that ensured each element for the user interface could be, 1; Painted onto the screen and, 2; Detected by the user's mouse. I created three main types: Button, Graphic and Text. I also used Polymorphism to create more specialized types of these components: Tab and TextBox. Lastly, I used combinations of these objects to create individual special components: Customer (delivery trucks) and Comment. Using this interface simplified and optimized the creation of my UI.

UI Components in the UI Class

```
public class UI {
    /** A list of images to be displayed on the user interf
    ArrayList<Graphic> graphic = new ArrayList<Graphic>();
    /** A list of buttons to be displayed on the user inter
    ArrayList<Button> button = new ArrayList<Button>();
    /** A list of texts to be displayed on the user interfa
    ArrayList<Text> text = new ArrayList <Text> ();
```

Sadly, you cannot create ArrayLists for objects of a particular interface, therefore I created a UI class to conveniently store, paint and check the three main types of UI Components used for just about every Game Screen.

Challenges and Accomplishments

Here is an honourable mention list of challenges conquered.

1. ***Organizing ingredients into one Ingredient class rather than having individual sub-classes for each ingredient.***

```
public class Ingredient {
    /** The index of the ingredient ...
    int index;
    /** The name of the ingredient ...
    String name;
    /** The unit the ingredient is mea
    String unit;
    /** The quantities and related cos
    double[][] store;
    /** How much of this ingredient is
    int stock;
    /** How much of this ingredient is
    int recipe;
    /** How does this ingredient run o
    String stockOut;
```

Convenient access to information about each Ingredient using a single class.

Other classes can now access ingredient information in simple for loops.

```
for(int i = 0; i <= 5; i++)
{
    ratings[i].rate = (int)math.map(abs(today.ingredient[i].recipe - optimalRecipe[i]), 5, 0, 0, 100);
}
```

Eg. Customer Class using a for loop to generate ratings on each ingredient.

2. ***Creating a MathTool Class to access a list of my own universally applicable mathematical functions for several classes and purposes.***

```
public class MathTool {

    /** Maps a position from one range into another range ...10 lines */
    public double map(double x, double imin, double imax, double fmin, double fmax) {...7 lines }

    /** Generates a random number in between the given range ...8 lines */
    public double random(int min, int max) {...3 lines }

    /** Selects a value according to a bell curve of probability ...10 lines */
    int gaussian(double pivot, double variance, double min, double max) {...11 lines }
}
```

Convenient access to commonly used mathematical functions to be used by other classes.

```
ratings[8] = new Rating (8, "rating", math.gaussian(math.map(today.rating, 0, 5, 0, 100), 2, 0, 100));
```

Eg. The Customer class using MathTool functions to determine its 0-100 rating for the shop's reputation.
Step 1 - The shop's 5 star rating uses map() to scale popular opinion 0-100.
Step 2 - gaussian() is used to create slight variance in the customer's opinion using a bell curve of probability.

3. ***Initiating each Customer's visit on its own independent Thread.***

The Day Class using Thread to initiate and update Customers throughout the day.

```
/**
 * Updates the customer's actions and information changing throu
 * day (views, rating, etc.).
 *
 * @param g Graphics object used to create graphics.
 */
public void paintCustomers(Graphics g) {
    rating = averageRating();
    for (int i = 0; i < customers.length; i++) {
        if (G.week[index].timeValue() > customers[i].time) {
            if (customers[i].getState() == Thread.State.NEW) {
                views++;
                customers[i].start();
            } else if (customers[i].willPurchase == true && cust
                customers[i].paint(g);
            }
        }
    }
}
```

Step 1. Goes through each customer.
Step 2. If the Customer's time has passed, but the Customer has not yet been initiated (Thread.State is New), then initiate the Customer.
Step 3. Update all customers who have already been initiated (using paint()).

4. ***Limiting the use of .png files to improve the speed of the program.***

```
/** Paints customer's out of five star rating ...8 lines */
public void paintStars(Graphics g, int x, int y, double r) {...11 lines }

/** The shape of a star to be painted an represent a Customer's rating on the ...15 lines */
public Shape createStar(double centerX, double centerY, double innerRadius, double outerRadius,
```

Eg. Drawing stars straight through paint component rather than using .png files so the program does not slow down.

5. ***Mapping the velocity of each delivery car/truck to slow down in front of the shop.***

```
velocity = abs(car.xpos - 200) / 10;
```

6. ***Sorting the customers by direction so all delivery cars driving in the front lane are painted first and using <u>recursion</u> to optimize sorting.***

```
/** Sorts the customer's based on which direction their delivery truck/car
public Customer[] quickSortCustomers(Customer[] arr, int low, int high) {..
```

7. ***Using trigonometric functions to animate snow falling, rain falling and clouds moving so there to optimize memory.***

```
public void paintRain(Graphics g, int x, int y, int w, int h)
{
    for(int i = 0; i < w/5; i ++)
    {
        if (cos(aCounter*20 + (i)) > 0) {
            g.setColor(dayNightColor(121, 144, 181));
        } else {
            g.setColor(new Color(0, 0, 0, 0));
        }
        g.fillOval(x + i*5, y + h/2 + (int) (sin(aCounter*20 + (i)) * h/2), 2, 10);
    }
}
```

Eg. Painting Rain
Only displaying rain drops when sin is increasing (when cos is positive) so all rain is travelling positively (downwards).

8. ***Giving customers their own slightly unique opinion based on a bell curve probability of chance.***

```
public int[] generateOptimalRecipe() {
    int[] oR = new int[today.ingredient.length];
    oR[0] = 1;
    oR[1] = math.gaussian(2, 0.25, 0, 10);
    oR[2] = math.gaussian(8, 0.25, 0, 10);
    oR[3] = math.gaussian(6, 0.25, 0, 10);
    oR[4] = math.gaussian(3, 0.25, 0, 10);
    oR[5] = math.gaussian(8, 0.25, 0, 10);
    return oR;
}
```

Eg. Each Customer uses gaussian() to have their own variated opinion on the optimal recipe worthy of a 5 star rating.

9. ***Using customer's individual ratings to determine how many views the page will get and how many viewers will purchase tea.***

Realistic Gameplay

```
public int customerNum() {

    int n = (int) (30 * weatherFactor());
    if (index > 0) {
        n = (int) ((G.week[index - 1].customers.length/G.wee
    }
    if (n < 10) {
        n = 10;
    }
    return n;
}
```

Generating the number of viewers based on the weather and the previous day's reviews.

```
public int purchaseChance() {
    int c = 0;
    for (int i = 7; i < ratings.length; i++)
    {
        c += ratings[i].rate;
    }
    return c / 4;
}
```

Generating the probability that the viewer will purchase tea based on their ratings for the price, weather and the current reviews.

```
return new Comment(comment, (double) ratings[6].rate, D);
```

Generating a review based on the customer's opinion of the recipe.

10. Writing and reading to a .txt file to remember the high score.

```
public Game() {
    try {
        File hs = new File("highscore.txt");
        if (hs.createNewFile()) {
            System.out.println("File created: " + hs.getName());
            highscore = 0;
            highscoreName = "";
        } else {
        }
        try {
            Scanner scanner = new Scanner(hs);
            String[] data;
            while (scanner.hasNextLine()) {
                data = scanner.nextLine().split(":");
                highscore = Double.parseDouble(data[1]);
                highscoreName = data[0];
            }
            scanner.close();
        } catch (FileNotFoundException e) {
            System.out.println("An error occurred with reading hi
            e.printStackTrace();
        }
    } catch (IOException e) {
        System.out.println("An error occurred with initiating hi
        e.printStackTrace();
    }
}
```

Reading high score from file upon starting up the program. If no such file exists, one will be automatically created.

Overwriting high score at the end of the game, if the current player has beaten the previous high score.

```
if (week[day].money > highscore) {
    highscore = week[day].money;
    highscoreName = userName;
    try {
        BufferedWriter writer = new BufferedWriter(new FileWriter("highscore.txt"));
        writer.write(userName + ":" + week[day].money);
        writer.close();
        System.out.println("Successfully wrote new highscore to the file.");
    } catch (IOException e) {
        System.out.println("An error occurred with recording new highscore.");
        e.printStackTrace();
```

Conclusion

If I could restart this program, I wouldn't make any major changes. Rather, I would love to expand and improve on what I already have (adding new flavours, hiring help, purchasing advertising and sound effects). Comparing this program to my Card Game, I can see how much I have learned and improved. I organize my code into specialized objects, take advantage of all the concepts learned throughout this course, simplify complex processes to use them repeatedly with ease, and take advantage of Java's built-in libraries and interfaces. Not only have I built a strong foundation of programming skills, but I have developed research and independent learning abilities. Reading through forums and articles has given me a better understanding of the language used by fellow programmers, allowing for easier collaboration in the future. I am very proud of my progress throughout these months, and I will continue to build upon this foundation of knowledge when attending university.