OOP Project#4 Report: ‘Among Us Run’ Game

[Team 13]

Department of Softwar[redacted]

Department of Software[redacted]

<Contents>

1. Brief project description
2. How to compile and execute
3. Description on functionality
4. How to implement important implementation issues
5. The result of UML modeling for system design
6. Execution results
7. Applying objected oriented concepts
8. Conclusion

1. Brief project description

This project implements a 2D running game using Java Swing. The game features a mechanism where the player moves left and right to avoid obstacle and collect coins.

2. How to compile and execute

Run IntelliJ IDEA 2023.3.5 on the Window OS(Window 11) and click the run button.

3. Description on functionality

1) Track class

This class manages and renders the tracks on which the player can move. It draws the tracks on the screen and positions obstacles and coins accordingly on the tracks.

2) Player class

This class manages the player character's state (position, lives, and coin count, animation) and handles movement and collision processing. It serves as player movement, life management, and coin collection.

3) Coin class

This class manages coins that the player can collect. It moves coins downward according to the game loop and checks for collisions with the player, increasing the player's coin count upon collision.

4) Obstacle class

This class manages obstacles that the player must avoid. It moves obstacles downward according to the game loop and checks for collisions with the player, reducing the player's lives upon collision.

5) RunningGame class

This class serves as the main game panel, rendering and updating all game elements. It acts as a setup for layout and image loading, game object initialization, tracks, obstacles and coins addition and background, timer, and input handling.

6) Main class

This class creates a simple GUI application using Java Swing and runs the game by adding a custom panel called ‘RunningGame’ to the frame.

4. How to implement important implementation issues

1) Track class

It uses ImageIcon to load the image from the specified path, converts it into an Image object, and stores it in the trackImage field to set the graphic image used for drawing the track on the screen. The draw() function draws the track image three times to arrange the tracks consecutively. This is used to create a scrolling effect for the tracks. The getX() function returns the track's x-coordinate, allowing other classes to access it.

2) Player class

It sets (this.currentTrack = 1) to ensure the player starts on the middle track. Through the ‘setPlayerImage()’ function, the player images are loaded from the specified path and each frame image is loaded in a loop. As a result, the ‘run1’, ‘run2’, ‘run3’, and ‘run4’ images are loaded and used as animation frames. After setting the player's size, (this.isAnimating = true) is set to activate the animation state, allowing the animation to proceed in the ‘update()’ method. Through the ‘update()’ method, the player character's animation frames are updated to ensure the animation continuously cycles, and the frame transitions occur every time ‘frameDelay’ is a multiple of 5. Through the ‘moveLeft()’ method, the player moves to the left track when the left arrow key is pressed and through the ‘moveRight()’method, the player moves to the right track when the right arrow key is pressed. In the ‘draw()’ method, the player is rendered on the screen using the ‘Graphics g’ object. When the player collides with an obstacle, the ‘hitObstacle()’ method is called to decrease lives, and when colliding with a coin, the ‘hitCoin()’ method is called to increase the coin count. The ‘getLives()’ and ‘getCoins()’ methods return the current number of lives and coins, respectively, also ‘resetLives()’ and ‘resetCoins()’ reset the lives and coins to their initial states when the game is restarted. The ‘resetPosition()’ method resets the player's position to the initial coordinates, and the ‘getBounds()’ method returns a ‘Rectangle’ object representing the player character's boundaries based on its current position and size for collision detection.

3) Obstacle class

In the ‘draw()’ method, the obstacle is rendered on the screen using the ‘Graphics g’ object. The ‘checkCollision()’ method retrieves the player's bounding ‘Rectangle’ and creates a new ‘Rectangle’ for the obstacle's boundaries to check if the two objects overlap. If they do, it returns ‘true’ otherwise, it returns ‘false’. The ‘getY()’ function returns the obstacle's y-coordinate, allowing other classes to access it. And the ‘setX()’ and ‘setY()’ methods set the obstacle's x and y coordinates, respectively.

4) Coin class

In the ‘draw()’ method, the coin is rendered on the screen using the ‘Graphics g’ object. The ‘checkCollision()’ method retrieves the player's bounding ‘Rectangle’ and creates a new ‘Rectangle’ for the coin's boundaries to check if the two objects overlap. If they do, it returns ‘true’ otherwise, it returns ‘false’. The ‘getY()’ function returns the coin's y-coordinate, allowing other classes to access it. And the ‘setX()’ and ‘setY()’ methods set the coin's x and y coordinates, respectively.

5) RunningGame class

Extends JPanel to enhace its functionality as a custom panel and implements ActionListener and KeyListener interfaces to handle action events and keyboard events.

In the constructor, the layout is set, and the images used in the game are loaded. The ‘startButton’, player, tracks, obstacles, coins, and background are initialized, and tracks, obstacles, and coins are added. A timer is set up to call the ‘actionPerformed’ method at 30-millisecond intervals and started, followed by setting up key input.

A ‘createButton()’ is created to generate custom buttons based on specified images, positions, and sizes, and it is called each time a button is created.

The ‘startButton()’ method creates the game start button using createButton() and sets an ActionListener so that clicking the button starts the game.

The ‘showPlayerSelection()’ method creates character selection buttons using createButton(), and when a button is clicked, it sets the player image and calls showConfirmButton().

The ‘showConfirmButton()’ method displays ‘Yes’ and ‘No’ buttons after character selection. It loads the "Would you like to play the game with this character" message image and creates the yes and no buttons using createButton() like the others. When the yes button is clicked, it set (isPlayerSelected = true) to indicate that character selection is complete and calls startGame() to begin the game. When the no button is clicked, it

sets (isPlayerSelected = false) to cancel character selection and allows re-selection.

The ‘showGameOver()’ method displays ‘Retry’ and ‘Exit’ buttons when the game ends. It loads the “Game Over” message image and creates the retry and exit buttons using createButton(). When the retry button is clicked, it calls resetGame() to initialize and restart the game, and when the exit button is clicked, it exits the program.

The ‘startGame()’ method, if (isGameStarted && isPlayerSelected) is true, removes all components from the panel, revalidates the layout, and repaints the screen.

The ‘resetGame()’ method initializes the game state flags and resets the player's lives, coins, and position. It then repositions obstacles and coins, removes all components from the panel, repaints the screen, and displays the start button again.

The ‘addObstacle()’ and ‘addCoin()’ methods are methods that place obstacles and coins at random positions on the tracks.

The ‘paintComponent()’ method draws appropriate graphics based on the current state of the game panel, including the game background, player, tracks, obstacles, coins, lives, and all images.

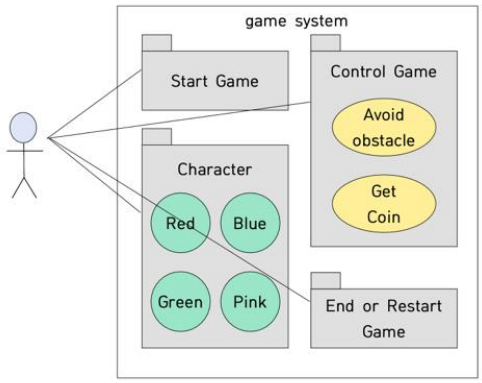
The ‘actionPerformed()’ method serves as the game loop. If (!isGameStarted || isGameOver), it exits the method. It repositions the background to implement background scrolling. It moves obstacles downwards, repositions them to new locations when they are out of the screen, and decreases the player's lives upon collision with obstacles. If lives reach 0 or below, it switches to game over state. Similarly, it moves coins downwards, repositions them when they are out of the screen, and increases the number of coins when colliding with the player.

The ‘keyPressed()’ method is called when the user presses a key. If the left arrow key is pressed, it moves the player to the left, and if the right arrow key is pressed, it moves the player to the right.

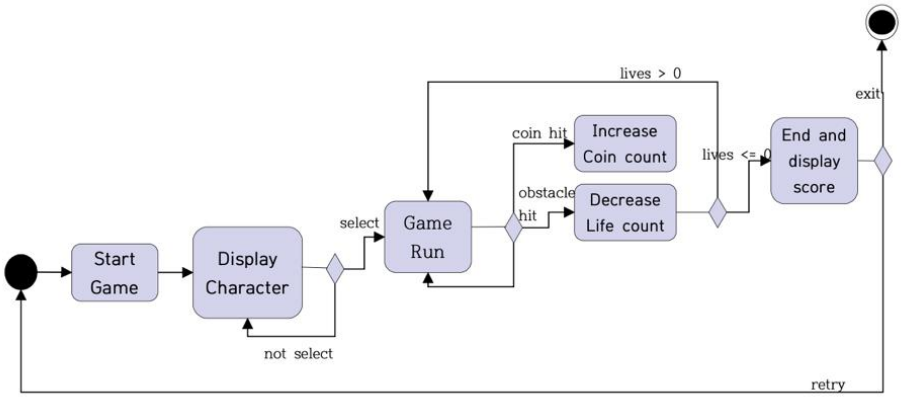
The ‘keyReleased()’ method is called when the user releases a key, stopping the player's movement.

5. The result of SW system design

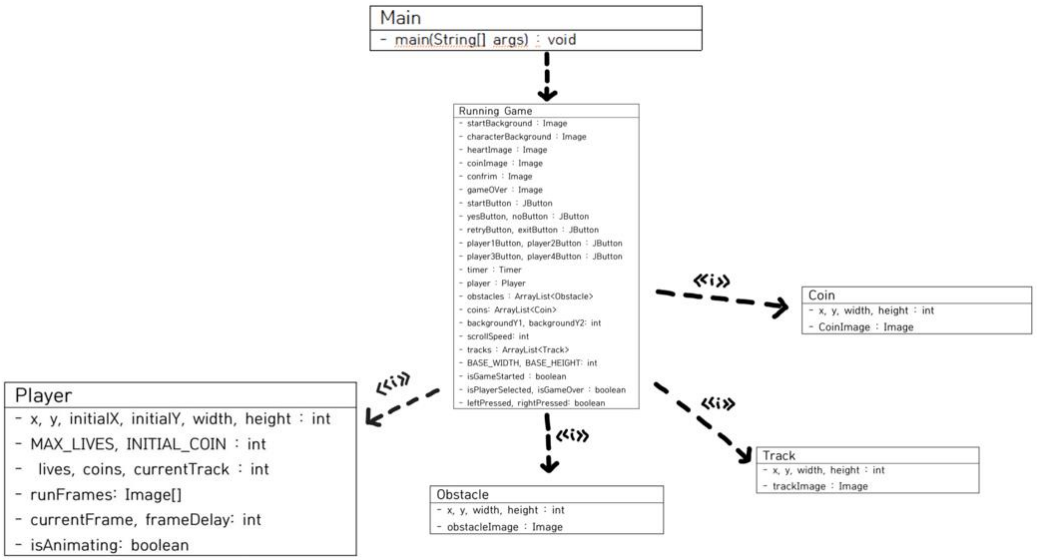
- case-use diagram



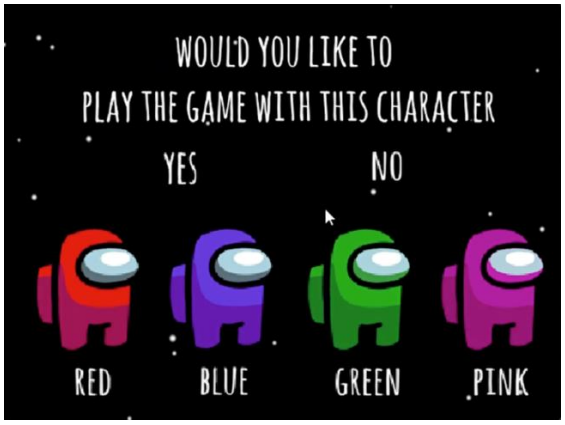
- activity diagram

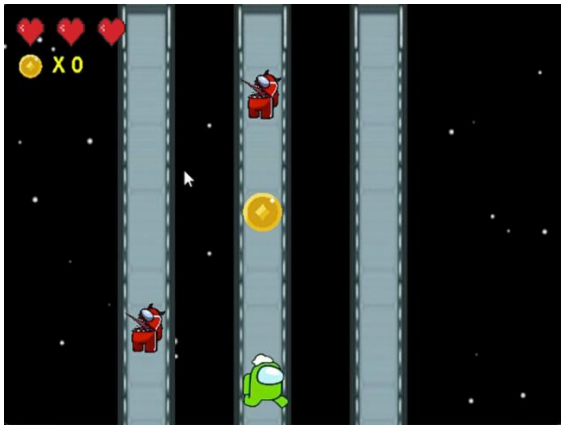


- class diagram



6. Execution results





7. Applying objected oriented concepts

1) Encapsulation

In Player, Obstacle, Coin, Track classes the encapsulate their respective properties (e.g., obstacleImage, coinImage, etc) and by using getter and setter methods (e.g., getX, getY) we can access to those properties.

2) Abstraction

By using abstraction, we simplified complex game object functions by modeling classes appropriate to the system's domain while exposing only essential features. The Player, Obstacle, Coin, and Track classes abstract specific entities of the game. In the Obstacle class abstracts the concept of obstacles, providing only the essential properties(x, y, width, height, obstacleImage) and methods (draw(), checkCollision()).

3) Inheritance

Running Game class extends JPanel, inheriting its graphical capabilities and functionalities, such as paintComponent() for rendering and event handling capabilities through the ActionListener and KeyListener interfaces.

4) Polymorphism

The Running Game class overrides the paintComponent() method from JPanel to customize how the game elements are drawn.

The actionPerformed() method from the ActionListener interface is implemented to handle timer events, allowing dynamic updates in the game loop.

The keyPressed() and keyRelaeased() methods from the keyListener interface are implemented to detect and process player input.

5) Modularity

The "Among us run" is broken into different classes:

- **Player** handles the player's state, actions and animation.
- **Obstacle** manages obstacles and their behavior when collided.
- **Coin** tracks collectible items.
- **Track** handles the background elements and scrolling logic.
- **Running Game** coordinates all components to create the game.

8. Conclusion

Through this project, we gained valuable insights into implementing various methods and classes, as well as designing user interfaces. Though we learned about UI in the previous course, by making our own game and working with image objects using Java's various given classes like Java Swing, we could learn how to handle graphics and manage animations effectively, which significantly improved our understanding of graphical programming.

Moreover, this project deepened our knowledge of object-oriented programming principles such as encapsulation, inheritance, and abstraction, and how they can be applied in real-world scenarios. It was difficult at first to handle the player and the obstacle interaction. But as we went further into developing our project, we addressed challenges like coin collecting, selecting character, thus enhancing our problem-solving skills.

If given more time, we would like to further implement some items to make the character change speed or increase their lives. Or we could insert sounds or animations when the character hits an obstacle/coin.