Programming Technology Documentation

Otar Jinchveladze

December 2024

Contents

1 Introduction

This document serves as the official documentation for the programming technology assignment. It outlines the goals and structure of developing as well as implementing the project. The primary objective of this assignment is to enhance our understanding of object-oriented programming principles. Additionally improve problem-solving skills through the design of a simulation-based game.

This document is created by Otar Jinchveladze, a second-year student pursuing computer science at Eötvös Loránd University (ELTE). This document will be reviewed by my teacher, Guettala Walid. He is a PhD student at ELTE with a background in deep learning and robotics, currently researching Graph Neural Networks.

Throughout this documentation, I will cover various aspects of the project. Including detailed descriptions of the exercise and the implementation of methods. Moreover, testing procedures and results will be provided to demonstrate the functionality and reliability of the program. All this presents an opportunity to gain hands-on experience in designing modular, maintainable code and applying core programming concepts in a practical scenario.

2 Description of the Exercise

2.1 Exercise 2

We have a rattlesnake in a desert, and our snake is initially two units long (head and rattler). We have to collect with our snake the foods on the level, that appears randomly. Only one food piece is placed randomly at a time on the level (on a field, where there is no snake). The snake starts off from the center of the level in a random direction. The player can control the movement of the snake's head with keyboard buttons. If the snake eats a food piece, then its length grow by one unit.

It makes the game harder that there are rocks in the desert. If the snake collides with a rock, then the game ends. We also lose the game, if the snake goes into itself, or into the boundary of the game level. In these situations show a popup messagebox, where the player can type his name and save it together with the amount of food eaten to the database. Create a menu item, which displays a highscore table of the players for the 10 best scores. Also, create a menu item which restarts the game.

3 Class Diagram

The class diagram for the Snake Game provides a comprehensive overview of the system architecture and the relationships between different components of the game. It is designed to illustrate the key classes, their attributes, methods, and the interactions among them.

Classes: Snake, Food, GameWindow, Rock, GamePanel (Subclass of JPanel)

3.1 Main Class Diagram

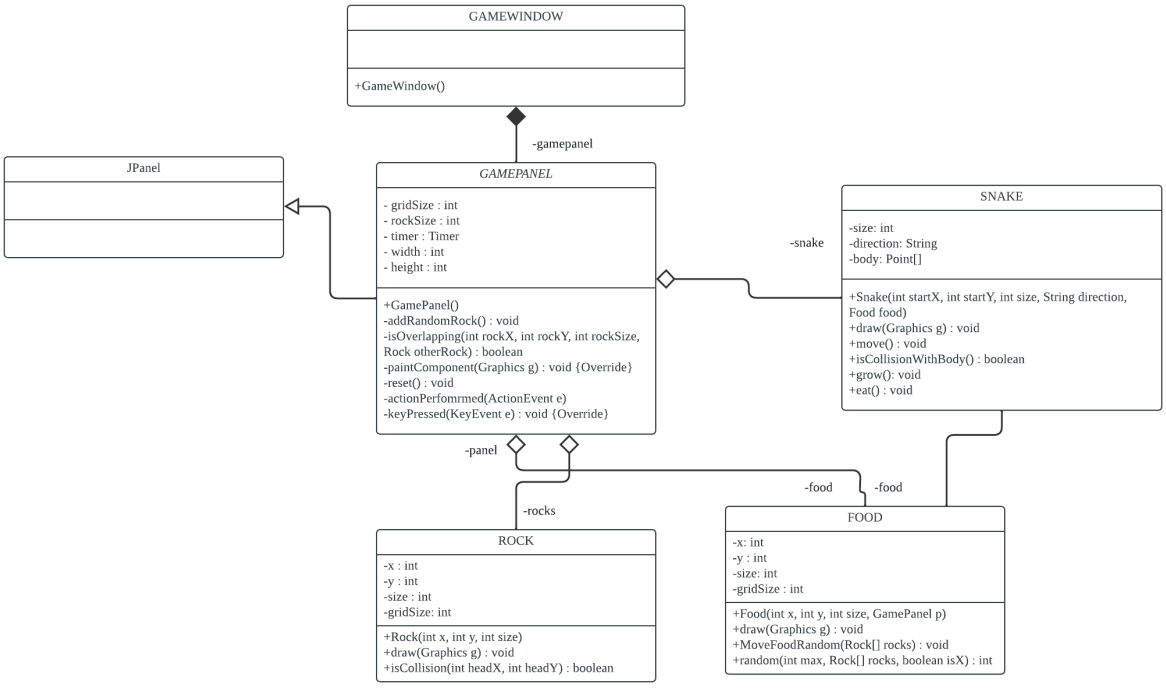

Figure 1: Class Diagram for the Tricky five-in-a-row game

4 Description of Methods

This documentation will not include getter and setter methods, as they are straightforward and commonly understood for managing access to private and protected fields. Getters and setters serve a clear purpose: getters allow external classes to access the values of private fields, while setters enable them to modify those values. Since these methods are standard practices in object-oriented programming and their functionality is generally familiar to developers, we will focus on documenting the more complex methods and functionality of the classes. This approach helps streamline the documentation by omitting repetitive explanations and allowing us to concentrate on the unique features and behaviors of the system that may require further clarification or context.

4.1 GamePanel Class

4.1.1 Method: GamePanel()

This is the constructor of the GamePanel class. It initializes the game components, including the snake (s), food (f), and a list of rocks. It sets up the game panel with a specific size and background color, then starts a game loop using a Timer to trigger regular updates.

4.1.2 Method: addRandomRock()

This method generates random positions for the rocks on the grid while ensuring that the rocks do not overlap with the snake, food, or other rocks. It uses a loop to keep generating positions until a valid one is found.

4.1.3 Method: paintComponent()

The paintComponent method is overridden to handle the drawing of the game elements on the screen. It draws the grid, the snake, the food, and the rocks. It ensures that the game elements are updated and displayed correctly with each repaint.

4.1.4 Method: isOverlapping()

This helper method checks if a rock's position overlaps with another rock's position by comparing their coordinates and sizes. It returns true if they overlap horizontally or vertically.

4.1.5 Method: reset()

The resetmethod resets the game state when the game is over. It repositions the snake, moves the food, clears the rocks, and prompts the player with a message asking if they want to play again. If the player chooses "yes," the game restarts; otherwise, the program exits.

4.1.6 Method: ActionPerformed()

This method handles the game loop's regular updates. It moves the snake, checks for collisions with the walls, body, or rocks, and checks if the snake eats the food. If a collision occurs, the game stops and

prompts the player with a "Game Over" message, followed by the reset method.

4.1.7 Method: KeyPressed()

This method captures key presses to change the snake's direction. It listens for the W, S, A, and D keys and updates the snake's direction accordingly.

4.2 Rock Class

4.2.1 Method: draw()

The draw method is responsible for visually rendering the rock on the game panel. It defines a triangular shape for the rock using arrays for x and y coordinates, setting the color to yellow, and finally drawing the polygon. This method ensures that each rock is displayed as a distinct, eye-catching obstacle within the game's grid.

4.2.2 Method: isCollision(Player winner)

This method checks if the snake's head coordinates intersect with the rock's area. It does so by defining a bounding box around the triangle (the rock) and verifying if the snake's head position falls within this box. If the head's coordinates are within the boundaries, it returns true, indicating a collision.

4.3 Food Class

4.3.1 Method: draw()

The draw method handles the graphical representation of the food on the game panel. It sets the color to red and draws the food as an oval at the specified coordinates with the given size. This visual representation makes the food clearly visible and distinct on the game grid.

4.3.2 Method: MoveFoodRandom()

This method repositions the food randomly within the game area. It calculates new coordinates for the food that do not overlap with any rocks. It uses a helper method random to ensure the food's new position avoids collisions with rocks, maintaining the game's challenge.

4.3.3 Method: random()

This helper method generates a random coordinate for the food either along the x or y axis (determined by isX). It ensures that the generated position does not overlap with any rocks, using a loop to test each generated position against the rocks' positions. The method returns a valid coordinate, ensuring that food placement respects the obstacles pr

4.4 Snake Class

4.4.1 Method: draw()

The draw method handles the graphical representation of the snake. It iterates over the snake's body to draw each segment, using a smaller size for the body segments compared to the head to create a visual distinction. The color green is used to make the snake stand out on the game panel.

4.4.2 Method: move()

This method updates the snake's position based on its current direction. It calculates the new position for the head and shifts the entire body accordingly. The last segment of the tail is removed, and the new head position is added at the beginning of the list, simulating movement.

4.4.3 Method: isCollisionWithBody()

This method checks if the snake's head collides with any part of its body, excluding the head itself. It returns true if such a collision is detected, indicating a game over condition.

4.4.4 Method: grow()

The grow method is used to increase the snake's length by duplicating the last segment's position and adding it to the body list. This method is typically called when the snake consumes food

4.4.5 Method: eat()

When the snake eats food, this method is called to trigger the MoveFoodRandom method of the Food class. This repositions the food on the game panel, ensuring it does not overlap with any rocks.

5 Connections between the events and event handlers.

5.0.1 GamePanel Class Event Handler

Event: Event: Pressing keys to change the snake's direction (W, A, S, D). .

Event Handler: The GamePanel class listens for key press events with a KeyListener. When a player presses one of the directional keys, the keyPressed (KeyEvent e) method is triggered, which updates the snake's direction using the setDirection (String direction) method based on the key pressed. This interaction allows real-time control of the snake during gameplay.

Event: The snake eats the food.

Event Handler: Within the actionPerformed (ActionEvent e) method, there is a check to see if the coordinates of the snake's head match those of the food. If they do, the eat (ArrayList¡Rock¿ rocks) method of the Snake class is called. This method leads to the snake growing and the food being repositioned randomly, avoiding the rocks, effectively updating the game state to reflect the snake's growth.

Event: The snake colliding with a rock or its own body.

Event Handler: This is handled within the actionPerformed (ActionEvent e) method, where checks are performed to determine if the snake's head coordinates intersect with any rock or its own body segments. If a collision is detected, the timer is stopped, the "Game Over!" message is displayed using JOptionPane, and the reset()method is called to restart or end the game based on the player's choice.

Event: Requesting to play again after the game is over.

Event Handler: After displaying the "Game Over!" message, the reset()method in GamePanel asks the player if they want to play again using a confirmation dialog (JOptionPane.showConfirmDialog). If the player chooses to continue, the game state is reset, and the timer is restarted, allowing the player to start a new game immediately. If not, the application exits.

6 Testing

Test Cases

1. **Test Initial Snake Position**
    - **As a:** creator
    - **I want to:** ensure the snake starts at the correct position
    - **Given:** the game panel is initialized
    - **When:** the game starts
    - **Then:** the snake's head is positioned at the specified start coordinates (e.g., 400, 300).
2. **Test Food Consumption**
    - **As a:** creator
    - **I want to:** ensure the snake grows after eating food
    - **Given:** the snake's head aligns with the food's position
    - **When:** the snake moves to the food's position
    - **Then:** the snake grows in length by one segment and the food is repositioned randomly avoiding rocks.
3. **Test Collision with Body**
    - **As a:** creator
    - **I want to:** ensure the game ends when the snake collides with its own body
    - **Given:** part of the snake's body is positioned in front of the head based on the current movement direction
    - **When:** the snake's head moves into one of its body segments
    - **Then:** the game stops, displays a "Game Over!" message, and offers the reset option.
4. **Test Collision with Rock**
    - **As a:** creator
    - **I want to:** ensure the game ends when the snake collides with a rock
    - **Given:** a rock is positioned in the snake's path
    - **When:** the snake's head moves into the rock
    - **Then:** the game stops, a "Game Over!" message is displayed, and the reset option is offered.
5. **Test Snake Movement**
    - **As a:** creator
    - **I want to:** verify that the snake moves correctly in all four directions
    - **Given:** the game is ongoing and the snake is not near the game boundaries or obstacles
    - **When:** directional keys (W, A, S, D) are pressed

- **Then:** the snake moves one grid size in the direction corresponding to the key pressed.

6. **Test Boundary Collision**
   - **As a:** creator
   - **I want to:** detect if the game ends when the snake hits the boundary
   - **Given:** the snake is positioned near the game panel boundary
   - **When:** the snake moves beyond the game panel boundary
   - **Then:** the game stops, a "Game Over!" message is displayed, and the reset option is presented.
7. **Test Game Reset**
   - **As a:** creator
   - **I want to:** ensure the game can be reset correctly
   - **Given:** the game has ended either by snake collision or boundary hit
   - **When:** the player chooses to restart the game from the "Game Over!" dialog
   - **Then:** the game restarts with the snake, food, and rocks reinitialized to their starting conditions.
8. **Test Direction Change Restriction**
   - **As a:** creator
   - **I want to:** ensure the snake cannot immediately reverse direction
   - **Given:** the snake is moving in a certain direction (e.g., "UP")
   - **When:** the opposite direction key (e.g., "DOWN") is pressed
   - **Then:** the direction does not change to the opposite, preventing the snake from colliding with itself.
9. **Test Random Food Placement**
   - **As a:** creator
   - **I want to:** ensure that food does not appear on rocks or outside the game boundaries
   - **Given:** the game is in progress and food needs to be repositioned after being eaten
   - **When:** the food is relocated
   - **Then:** the new position of the food is not overlapping with rocks or outside the game area.
10. **Test Rock Generation**
    - **As a:** creator
    - **I want to:** ensure rocks are placed randomly without overlapping each other, the snake, or the food
    - **Given:** the game starts or is reset
    - **When:** rocks are generated on the game panel
    - **Then:** no rocks overlap with each other, the initial snake position, or the food, and are within the game boundaries.