EÖTVÖS LORÁND UNIVERSITY
INFORMATICS FACULTY
PROGRAMMING TECHNOLOGY COURSE
(Template belongs to Vilnius University)

Snake Game

Documentation of the Assignment No. 3, Task 2, Version 2

Author: Monika Mirbakaite (YS0EG6)

ELTE email: ys0eg6@inf.elte.hu

Evaluated by: Lecturer Szendrei Rudolf,
Instructor Török Zoltán Ákos

Budapest
2025

0. CHANGES MADE SINCE VERSION 1

0.0. Source Code

1. More clear folder structure that shows project's MVC practices
2. A new class "GameConfig" in folder "view" that is responsible for dimensions
3. A new class in folder "model" "AppleManager" that is responsible for the food creation and "GameLogic" now holds only the logic of the game
4. In the class "GameController" updated "handleKeyPress" method, so the game would work only if correct keys are pressed and otherwise game is over
5. Fixed timer so it would run regardless of player is pressing keys or not

0.1. Testing

6. Updated White Box test so it would work with the updated source code

0.2. Documentation

7. An updated class Diagram

In the section

Picture 1. A Class Diagram of the Game.

8. implementation only the most important methods are shown.
9. Updated White Box test section

1. DESCRIPTION

1.1. Common requirements

1. Your program should be user friendly and easy to use. You have to make an object oriented implementation, but it is not necessary to use multilayer architectures (MV, MVC etc.).
2. You have to use simple graphics for the game display. The "sprite" of the player's character should be able to moved with the well-known WASD keyboard buttons. You can also implement mouse event handlers to other functions of the game.
3. You can generate the game levels with an algorithm, or simply load them from files. If you load the levels from file, then each one should be put into different files, and you have to create at least 10 predefined levels. If you generate the levels, then take care that the levels should be playable (player should be able to solve it).
4. Each game needs to have a timer, which counts the elapsed time since the start of the game level.
5. The documentation should contain the description of the task, its analysis, the structure of the program (UML class diagram), a chapter for the implementation, which describes the most interesting algorithms (e.g. that generates the level) etc. of the program. Also, don't forget to show the connections between the events and their handlers.
6. The task description should contain the minimal requirements. It is free to add new functionalities to the games.

1.2. Exercise

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 message box, 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.

2. ANALYSIS

2.1. Task Analysis (Key Points)

2.1.1. Game Elements

- A rectangular grid representing the desert where the game is played.
- A snake consisting of a head and tail segments, initially two units long.
- Food pieces that appear one at a time on empty tiles for the snake to consume.
- Rocks placed on the grid as obstacles.
- A "Game Over" popup message that appears when the player loses.
- A menu containing options to restart the game and view a high score table.
- A high score table that records the top 10 scores, along with player names.

2.1.2. Game Flow

- The snake starts in the middle of the grid with a fixed initial direction.
- The player uses the WASD keys to control the movement of the snake.
- A single food piece spawns on a random empty tile; when eaten, the snake grows by one unit.
- If the snake collides with a rock, itself, or the boundary of the grid, the game ends, displaying the "Game Over" popup.
- The player can enter their name and save their score (based on the number of food pieces eaten) to the database.
- The player can restart the game from the menu or view the high score table.

2.2. Solution Plan (Implementation Steps)

2.2.1. Design Class Diagram

- Define the main components: Snake, Food, Rock, Grid, GameController, GUIHandler, and HighScoreManager.
- Map out the interactions between these classes (e.g., how GameController manages the game state and GUIHandler updates the display).

2.2.2. Implement Classes

- Create the Snake class to manage movement, growth, and collision detection.
- Create the Food class to handle food placement and consumption.
- Create the Rock class to define obstacles.
- Create the GameController class to manage the overall game logic.
- Create the HighScoreManager class to manage saving and retrieving scores.
- Create the GUIHandler class to manage rendering the game and handling user input.

2.2.3. Assign Responsibilities

- Define how each class contributes to gameplay.

2.2.4. Develop the GUI

- Implement a simple graphical interface to display the grid, snake, food, and rocks.
- Add functionality for the WASD keys to control the snake.
- Include the menu for restarting the game and viewing high scores.

2.2.5. Integrate Event Handlers

- Link keyboard input to snake movement.
- Handle collisions and game-over conditions with appropriate event responses.
- Connect the menu options to their respective actions (restart game, display high scores).

2.2.6. Test Gameplay

Ensure that:

- The snake moves correctly and grows when eating food.
- Food spawns in valid locations.
- Rocks are placed properly and block the snake.
- Collision detection works for all conditions (snake-self, snake-rock, snake-boundary).
- High scores save and display correctly.

2.2.7. Perform User Testing

- Test the game for usability, ensuring smooth controls and clear instructions.
- Verify that the menu options and high score functionality are intuitive and reliable.

3. CLASS DIAGRAM

In **Error! Reference source not found.** a class diagram of Snake game is displayed:

- SnakeGame has exactly one GameController, which it uses to coordinate the game.

SnakeGame depends on GameController because it creates an instance of GameController to manage the overall game logic and flow. This relationship is a composition.

- One GameController has one GameLogic and one GamePanel.

The GameController depends on both GameLogic and GamePanel to function. GameLogic handles the game's rules and mechanics, while GamePanel manages the graphical user interface. Both are created and managed by GameController, making this a composition.

- GameLogic works with exactly one LevelGenerator.

The LevelGenerator is used to create levels and obstacles for the game. GameLogic does not own LevelGenerator directly, meaning LevelGenerator can exist independently, making this an aggregation.

- GameLogic also works with one DatabaseManager.

The DatabaseManager is responsible for saving and retrieving high scores. Like LevelGenerator, DatabaseManager is not tightly bound to GameLogic and can exist independently. This makes the relationship an aggregation.

- GameLogic depends on exactly one GameConfig to obtain game settings.

The GameConfig provides configuration details such as screen width, height, and unit size. GameConfig is not created or owned by GameLogic, making this an association.

- GameLogic depends on one AppleManager to handle apple placement.

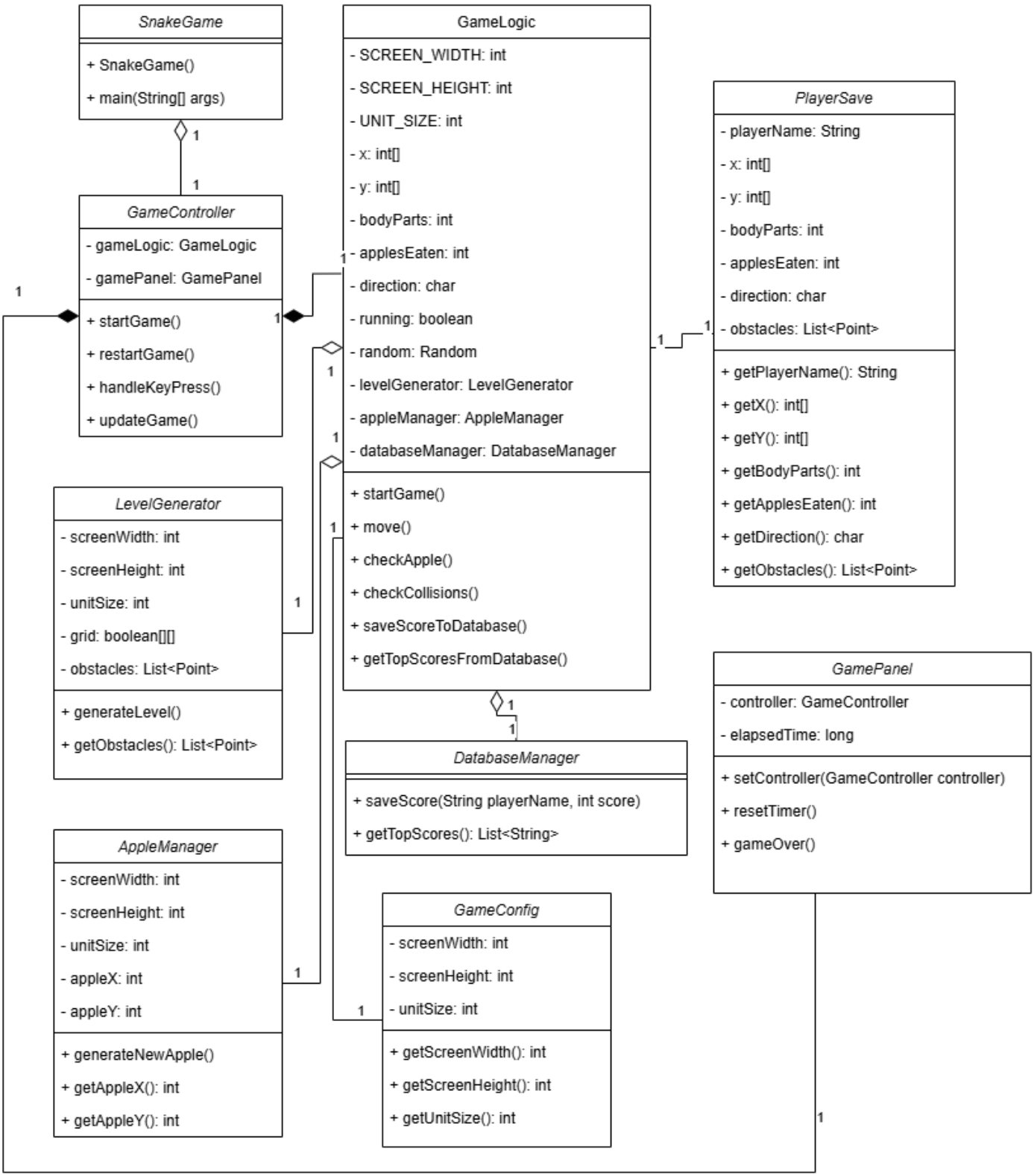
AppleManager is responsible for generating and managing the apple's position. It is created and tightly managed by GameLogic, making this a composition.

- PlayerSave references GameLogic to save the game's state.

PlayerSave depends on GameLogic because it captures its attributes, such as the snake's position, direction, and the game's obstacles. However, GameLogic is not created or managed by PlayerSave, making this an association.

- One GamePanel depends on GameController.

The GamePanel relies on the GameController to process user input and update the game's state. GamePanel cannot function independently of GameController because it relies on its methods to handle key presses and update the display. This relationship is an aggregation.



Picture 1. A Class Diagram of the Game.

4. IMPLEMENTATION

4.1. Package ‘game.controller’

4.1.1. Class ‘GameController’

`handleKeyPress(KeyEvent e)`. Handles user input for controlling the snake's direction. Ensures the snake cannot reverse into itself. Ends the game if an invalid key is pressed.

```
public void handleKeyPress(KeyEvent e) {
    char keyChar = e.getKeyChar();
    char direction = gameLogic.getDirection();

    switch (keyChar) {
        case 'a' -> {
            if (direction != 'R') gameLogic.setDirection('L');
        }
        case 'd' -> {
            if (direction != 'L') gameLogic.setDirection('R');
        }
        case 'w' -> {
            if (direction != 'D') gameLogic.setDirection('U');
        }
        case 's' -> {
            if (direction != 'U') gameLogic.setDirection('D');
        }
        default -> {
            gamePanel.gameOver();
            return;
        }
    }
    updateGame();
}
```

`updateGame()`. Updates the game state by moving the snake, checking for collisions, and repainting the UI. Detects if the snake eats the apple or hits an obstacle.

```
public void updateGame() {
    gameLogic.move();
    if (gameLogic.checkApple()) {
        gamePanel.repaint();
    }
    if (gameLogic.checkCollisions()) {
        gamePanel.gameOver();
    }
    gamePanel.repaint();
}
```

4.2. Package ‘game.model’

4.2.1. Class ‘AppleManager’

`generateNewApple(List<Point> obstacles, int[] snakeX, int[] snakeY, int bodyParts)`. Generates a new apple position on the grid, avoiding collisions with the snake and obstacles.

```
public void generateNewApple(List<Point> obstacles, int[] snakeX, int[] snakeY, int
bodyParts) {
    boolean validPosition;
    do {
        validPosition = true;
        appleX = random.nextInt(screenWidth / unitSize) * unitSize;
        appleY = random.nextInt(screenHeight / unitSize) * unitSize;
```

```
        // Check if apple overlaps with obstacles
        for (Point obstacle : obstacles) {
            if (appleX == obstacle.x * unitSize && appleY == obstacle.y * unitSize) {
                validPosition = false;
                break;
            }
        }

        // Check if apple overlaps with the snake
        for (int i = 0; i < bodyParts; i++) {
            if (appleX == snakeX[i] && appleY == snakeY[i]) {
                validPosition = false;
                break;
            }
        }
    } while (!validPosition);
}
```

4.2.2. Class ‘DatabaseManager’

`saveScore(String playerName, int score)`. Saves the player's score to a MySQL database for leaderboard tracking.

```
public void saveScore(String playerName, int score) {
    String query = "INSERT INTO Highscores (player_name, score) VALUES (?, ?)";
    try (Connection connection = DriverManager.getConnection(DB_URL, USERNAME,
PASSWORD);
         PreparedStatement preparedStatement = connection.prepareStatement(query)) {

        preparedStatement.setString(1, playerName);
        preparedStatement.setInt(2, score);
        preparedStatement.executeUpdate();
        System.out.println("Score saved successfully!");

    } catch (SQLException e) {
        throw new RuntimeException("Failed to save score to database", e);
    }
}
```

`getTopScores()`. Retrieves the top 10 scores from the database, formatted with player name, score, and timestamp.

```
public List<String> getTopScores() {
    String query = "SELECT player_name, score, timestamp FROM Highscores ORDER BY
score DESC LIMIT 10";
    List<String> topScores = new ArrayList<>();

    try (Connection connection = DriverManager.getConnection(DB_URL, USERNAME,
PASSWORD);
         Statement statement = connection.createStatement();
         ResultSet resultSet = statement.executeQuery(query)) {

        while (resultSet.next()) {
            String scoreEntry = String.format("%s - %d points (%s)",
                    resultSet.getString("player_name"),
                    resultSet.getInt("score"),
                    resultSet.getString("timestamp"));
            topScores.add(scoreEntry);
        }

    } catch (SQLException e) {
        throw new RuntimeException("Failed to fetch scores from database", e);
    }

    return topScores;
```

```
}
```

4.2.3. Class ‘GameLogic’

`startGame()`. Initializes the snake, resets the score, and generates the initial level and apple.

```
public void startGame() {
    bodyParts = 2;
    applesEaten = 0;
    direction = 'R';
    running = true;

    for (int i = 0; i < bodyParts; i++) {
        x[i] = SCREEN_WIDTH / 2 - (i * UNIT_SIZE);
        y[i] = SCREEN_HEIGHT / 2;
    }

    levelGenerator.generateLevel();
    appleManager.generateNewApple(levelGenerator.getObstacles(), x, y, bodyParts);
}
```

`move()`. Moves the snake in the current direction, updating the position of each body part.

```
public void move() {
    for (int i = bodyParts; i > 0; i--) {
        x[i] = x[i - 1];
        y[i] = y[i - 1];
    }
    switch (direction) {
        case 'U' -> y[0] -= UNIT_SIZE;
        case 'D' -> y[0] += UNIT_SIZE;
        case 'L' -> x[0] -= UNIT_SIZE;
        case 'R' -> x[0] += UNIT_SIZE;
    }
}
```

`checkApple()`. Checks if the snake has eaten an apple. If true, increases the body size, increments the score, and generates a new apple.

```
public boolean checkApple() {
    if (x[0] == appleManager.getAppleX() && y[0] == appleManager.getAppleY()) {
        bodyParts++;
        applesEaten++;
        appleManager.generateNewApple(levelGenerator.getObstacles(), x, y, bodyParts);
        return true;
    }
    return false;
}
```

`checkCollisions()`. Detects collisions with the snake’s body, walls, or obstacles, and ends the game if a collision occurs.

```
public boolean checkCollisions() {
    // Check collision with itself
    for (int i = bodyParts; i > 0; i--) {
        if (x[0] == x[i] && y[0] == y[i]) {
            running = false;
            return true;
        }
    }

    // Check collision with walls
    if (x[0] < 0 || x[0] >= SCREEN_WIDTH || y[0] < 0 || y[0] >= SCREEN_HEIGHT) {
        running = false;
        return true;
```

```
    }

    // Check collision with obstacles
    for (Point obstacle : levelGenerator.getObstacles()) {
        if (x[0] == obstacle.x * UNIT_SIZE && y[0] == obstacle.y * UNIT_SIZE) {
            running = false;
            return true;
        }
    }

    return false;
}
```

saveScoreToDatabase(String playerName). Saves the current player's score to the database.

```
public void saveScoreToDatabase(String playerName) {
    databaseManager.saveScore(playerName, applesEaten);
}
```

getTopScoresFromDatabase(). Fetches and returns the top scores from the database.

```
public List<String> getTopScoresFromDatabase() {
    return databaseManager.getTopScores();
}
```

4.2.4. Class ‘LevelGenerator’

generateLevel(). Randomly generates obstacles for the game level. Ensures a variety of layouts between plays.

```
public void generateLevel() {
    obstacles.clear();
    for (boolean[] row : grid) {
        Arrays.fill(row, false);
    }

    int obstacleCount = random.nextInt(10) + 5; // Generate between 5 and 14 obstacles
    for (int i = 0; i < obstacleCount; i++) {
        int obstacleX = random.nextInt(screenWidth / unitSize);
        int obstacleY = random.nextInt(screenHeight / unitSize);

        obstacles.add(new Point(obstacleX, obstacleY));
        grid[obstacleX][obstacleY] = true;
    }
}
```

4.3. Package ‘game.view’

4.3.1. Class ‘GamePanel’

gameOver(). Handles the end-of-game process, including saving the score, displaying the leaderboard, and offering a restart option.

```
public void gameOver() {
    timer.stop();
    String playerName = JOptionPane.showInputDialog(this, "Enter Your Name:", "Game
Over", JOptionPane.PLAIN_MESSAGE);
    if (playerName == null || playerName.trim().isEmpty()) {
        playerName = "Unknown Player";
    }

    saveAndDisplayLeaderboard(playerName);
```

```
    int choice = JOptionPane.showConfirmDialog(this, "Do you want to restart?",
"Restart Game", JOptionPane.YES_NO_OPTION);
    if (choice == JOptionPane.YES_OPTION) {
        elapsedTimeInSeconds = 0; // Reset the timer
        timer.start(); // Restart the timer
        controller.restartGame();
    } else {
        System.exit(0);
    }
}
```

`draw(Graphics g)`. Draws the game elements (snake, apple, obstacles, score, timer) dynamically based on the game state.

```
private void draw(Graphics g) {
    GameLogic gameLogic = controller.getGameLogic();

    if (gameLogic.isRunning()) {
        drawObstacles(g, gameLogic);
        drawApple(g, gameLogic);
        drawPlayer(g, gameLogic);
        drawScoreAndTimer(g, gameLogic);
    } else {
        gameOver();
    }
}
```

`saveAndDisplayLeaderboard(String playerName)`. Saves the player's score and displays the top leaderboard using a dialog.

```
private void saveAndDisplayLeaderboard(String playerName) {
    GameLogic gameLogic = controller.getGameLogic();
    gameLogic.saveScoreToDatabase(playerName);

    List<String> topScores = gameLogic.getTopScoresFromDatabase();
    StringBuilder scoresDisplay = new StringBuilder("Top Scores:\n");
    for (String score : topScores) {
        scoresDisplay.append(score).append("\n");
    }
    JOptionPane.showMessageDialog(this, scoresDisplay.toString(), "Top Scores",
JOptionPane.INFORMATION_MESSAGE);
}
```

4.4. Package ‘game’

4.4.1. Class ‘SnakeGame’

`main(String[] args)`. The entry point for the game, initializing the game configuration, logic, controller, and UI.

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(SnakeGame::new);
}
```

4.5. DataGrip SQL Queries

```
create database snake2;

USE snake2;

CREATE TABLE Highscores (
    id SERIAL PRIMARY KEY,
    player_name VARCHAR(50),
    score INT,
```

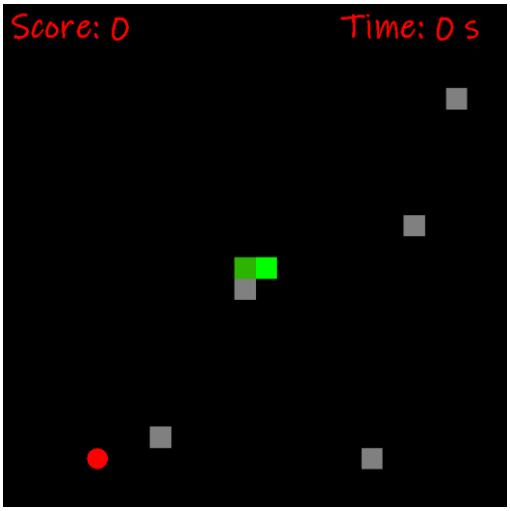
```
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

SELECT * FROM Highscores;

SELECT * FROM Highscores
ORDER BY score DESC
LIMIT 10;

DROP TABLE Highscores;

DROP DATABASE snake2;
```

TEST REPORT

4.6. Black Box (Running Application)

4.6.1. Starting Page

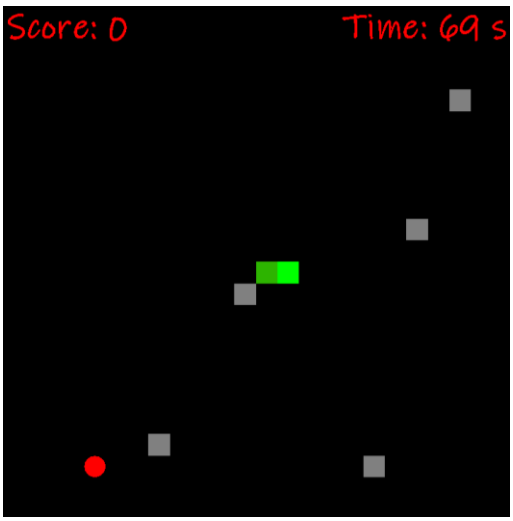
In Picture 2 the start of the game is displayed. Randomly set level (rocks), snake in the middle, the food - an apple (one at a time), timer, and score.



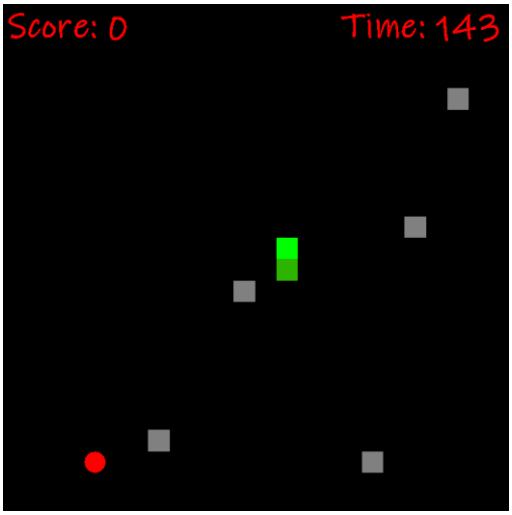
Picture 2. The Start.

4.6.2. WASD Buttons

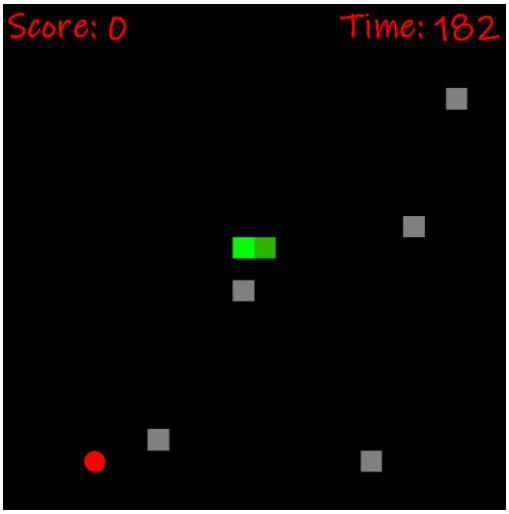
In Picture 3, Picture 4, Picture 5, Picture 6 the snake movements are displayed after using W, A, S, D buttons individually.



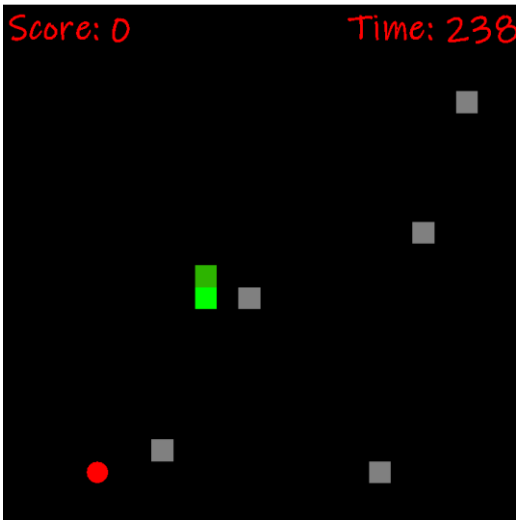
Picture 3. 'D' button was pressed.



Picture 4. 'W' button was pressed.



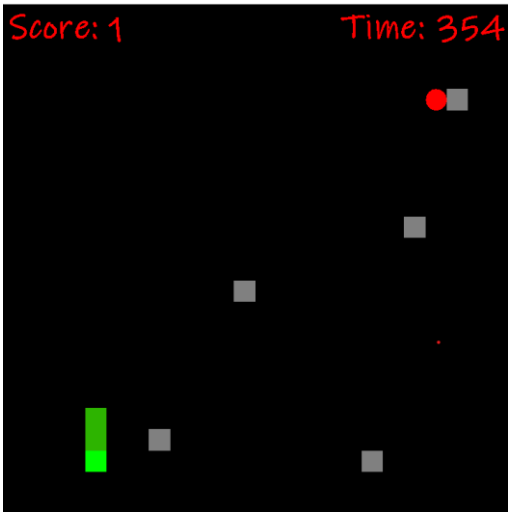
Picture 5. 'A' button was pressed.



Picture 6. 'S' button was pressed.

4.6.3. Eating An Apple

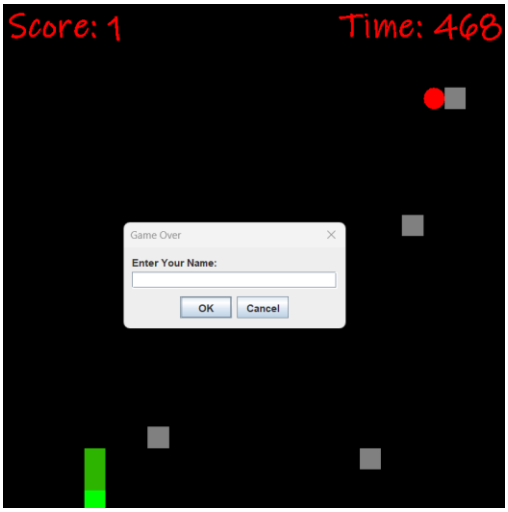
In Picture 7, a snake after eating an apple is displayed. The snake is longer by 1 unit, and the apple appears in the another spot of the board.



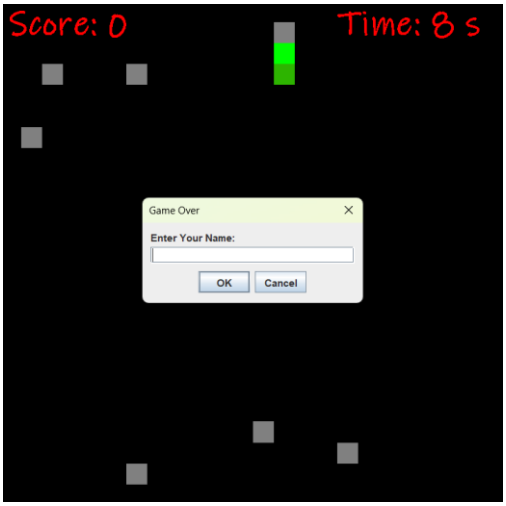
Picture 7. Progress in the game.

4.6.4. Loosing

In Picture 8 loosing is displayed after hitting a border of the board. Picture 9 loosing is displayed after hitting a rock. Both ways user now has to enter their name. If they enter it – their name will appear in the top scores table and in the result table. If they press ‘cancel’ or ‘ok’ (without entering a name) this user will be set as “Uknown user’.



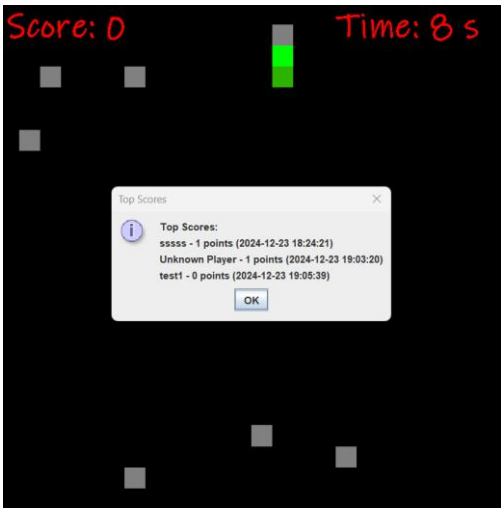
Picture 8. User Lost (1)



Picture 9. User Lost (2)

4.6.5. Entering a name

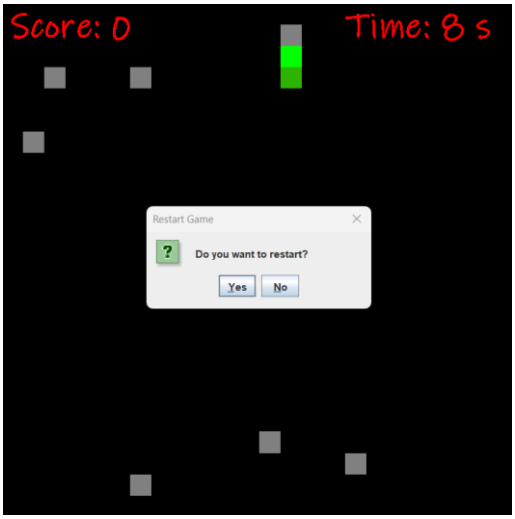
In Picture 10 we can the top 10 top scores.



Picture 10. Top Scores.

4.6.6. Exiting/New Game

In Picture 11 the final menu window is displayed. If a user presses ‘Yes’ they can play again from the start. If a user presses ‘No’ they exit from the game.



Picture 11. Restart/End.

4.6.7. Results on DataGrip

In Picture 12 result table on DataGrip is displayed.

	id	player_name	score	timestamp
1	1	sssss	1	2024-12-23 18:24:21
2	2	Unknown Player	1	2024-12-23 19:03:20
3	3	test1	0	2024-12-23 19:05:39

Picture 12. Result Table.

4.7. White Box

4.7.1. Package ‘testing’, Class ‘GameLogicTest.java’

```
package testing;

import game.model.GameLogic;
import game.model.DatabaseManager;
import game.model.*;
import game.view.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.awt.*;
import java.util.ArrayList;
import java.util.List;

import static org.junit.jupiter.api.Assertions.*;

/**
 * This class provides white-box testing for the game logic in the {@code GameLogic}
class.
 */
class GameLogicTest {

    private GameLogic gameLogic;
```

```
    private GameConfig config;

    /**
     * Sets up a new {@code GameLogic} instance before each test.
     */
    @BeforeEach
    void setUp() {
        config = new GameConfig(600, 600, 25);
        gameLogic = new GameLogic(config);
    }

    /**
     * Tests the initialization of the game to ensure the snake starts with the
correct
     * number of body parts and the game is running.
     */
    @Test
    void testGameInitialization() {
        gameLogic.startGame();

        assertEquals(2, gameLogic.getBodyParts());
        assertEquals('R', gameLogic.getDirection());
        assertTrue(gameLogic.isRunning());
    }

    /**
     * Tests that a new apple does not spawn on an obstacle.
     */
    @Test
    void testAppleDoesNotSpawnOnObstacle() {
        gameLogic.startGame();

        List<Point> obstacles = gameLogic.getObstacles();
        gameLogic.checkApple(); // Generates the new apple

        Point apple = new Point(gameLogic.getAppleX() / config.getUnitSize(),
                gameLogic.getAppleY() / config.getUnitSize());

        System.out.println("Obstacles: " + obstacles);
        System.out.println("Apple: " + apple);

        assertFalse(obstacles.contains(apple), "Apple should not spawn on an
obstacle.");
    }


    /**
     * Tests the {@code checkApple} method to verify that the snake's body grows
     * and the score increases when an apple is eaten.
```

```
     */
    @Test
    void testAppleConsumptionIncreasesBodyAndScore() {
        gameLogic.startGame();

        int initialBodyParts = gameLogic.getBodyParts();
        int initialApplesEaten = gameLogic.getApplesEaten();

        // Set the apple's position directly
        gameLogic.getX()[0] = gameLogic.getAppleX();
        gameLogic.getY()[0] = gameLogic.getAppleY();

        boolean ateApple = gameLogic.checkApple();

        System.out.println("Initial Body Parts: " + initialBodyParts);
        System.out.println("Initial Apples Eaten: " + initialApplesEaten);
        System.out.println("Ate Apple: " + ateApple);
        System.out.println("New Body Parts: " + gameLogic.getBodyParts());
        System.out.println("New Apples Eaten: " + gameLogic.getApplesEaten());

        assertTrue(ateApple, "checkApple should return true if the apple is
eaten.");
        assertEquals(initialBodyParts + 1, gameLogic.getBodyParts(), "Body parts
should increase by 1 after eating an apple.");
        assertEquals(initialApplesEaten + 1, gameLogic.getApplesEaten(), "Apples
eaten should increase by 1 after eating an apple.");
    }


    /**
     * Tests the {@code saveScoreToDatabase} method to verify it correctly
interacts
     * with the {@code DatabaseManager}.
     */
    @Test
    void testScoreSavingToDatabase() {
        class MockDatabaseManager extends DatabaseManager {
            private final List<String> savedScores = new ArrayList<>();

            @Override
            public void saveScore(String playerName, int score) {
                savedScores.add(playerName + ": " + score);
            }

            public boolean isScoreSaved(String expected) {
                return savedScores.contains(expected);
            }
        }
```

```
            MockDatabaseManager mockDatabaseManager = new MockDatabaseManager();

            GameLogic gameLogicWithMock = new GameLogic(config) {
                @Override
                public void saveScoreToDatabase(String playerName) {
                    mockDatabaseManager.saveScore(playerName, getApplesEaten());
                }
            };

            String playerName = "TestPlayer";
            gameLogicWithMock.saveScoreToDatabase(playerName);

            assertTrue(mockDatabaseManager.isScoreSaved("TestPlayer: 0"));
        }

        /**
         * Tests the {@code checkCollisions} method to verify that the game ends
correctly
         * when the snake collides with itself.
         */
        @Test
        void testSnakeCollisionEndsGame() {
            gameLogic.startGame();

            gameLogic.getX()[0] = gameLogic.getX()[1];
            gameLogic.getY()[0] = gameLogic.getY()[1];

            boolean collision = gameLogic.checkCollisions();

            assertTrue(collision);
            assertFalse(gameLogic.isRunning());
        }
    }
```

5. USER STORIES

1. As a player, I want to start a new game from a main menu so that I can play the rattlesnake game.
2. As a player, I want to use the WASD keys to control the snake's movement so that I can guide the snake to eat food and avoid obstacles.
3. As a player, I want the game to randomly generate a food piece on the map at a time so that I can collect it and grow the snake.
4. As a player, I want the game to place rocks on the map as obstacles so that it becomes more challenging to navigate.
5. As a player, I want the game to end if the snake collides with itself, the map boundaries, or a rock so that it feels realistic and challenging.
6. As a player, I want to start the game with a snake that is two units long so that I can progressively grow the snake by eating food.
7. As a player, I want the snake to grow by one unit each time it eats food so that the challenge increases as the game progresses.
8. As a player, I want the game to display a "Game Over" popup when I lose so that I know why the game ended.
9. As a player, I want to enter my name and save my score when I lose so that I can see my progress in the high score table.
10. As a player, I want to view a high score table in the menu so that I can compare my performance with others.
11. As a developer, I want the game to store high scores in a database so that the leaderboard data persists between game sessions.
12. As a player, I want a menu option to restart the game so that I can quickly play again without closing and reopening the application.
13. As a player, I want the snake to start at a fixed position so that I can consistently begin the game in a familiar way.
14. As a player, I want to see my score displayed during the game so that I know how well I'm doing.
15. As a player, I want the game to provide multiple levels so that I have new challenges to play through.