Universitatea *Transilvania* din Brașov
Facultatea de Matematică și Informatică
Departamentul de Matematică și Informatică

Documentaţie proiect JAVA
Maze Runner
Grupa 10LF333
Vîlcu Andreea

1 Introducere şi obiective

1.1 Descriere generală

MazeRunner este o aplicaţie implementata în limbajul de programare Java, în care jucătorii sunt provocați să navigheze prin labirinturi generate procedural, cu scopul de a găsi calea optimă către ieșire. Acest proiect oferă o experiență interactivă de joc, punându-se accentul pe algoritmi de generare de labirinturi, optimizarea mișcării jucătorului și implementarea unei interfețe grafice intuitive.

Labirintul este generat procedural folosind algoritmi de tipul DFS (Depth-First Search), iar jocul include mecanisme de colectare a power-up-urilor și de gestionare a scorurilor.

1.2 Obiective Principale

- **Implementarea unui generator procedural de labirinturi:** Folosirea unui algoritm de generare a labirinturilor care creează un labirint unic de fiecare dată, pe baza dimensiunii și dificultății alese de jucător. Există trei niveluri de dificultate care afectează densitatea pereților: **ușor - 20% pereți, mediu - 35% pereți, dificil - 50% pereți**. De asemenea algoritmul asigură existența unui drum între punctele de start și ieșire.
- **Crearea unui sistem de scoring și clasamente:** Păstrarea scorurilor jucătorilor și înregistrarea acestora în fișiere pentru compararea performanțelor.
- **Dezvoltarea unei interfețe grafice intuitive:** Crearea unui GUI (Graphical User Interface) ușor de utilizat care să permită jucătorului să interacționeze cu jocul.
- **Salvarea persistentă a datelor:** Implementarea unui sistem de salvare care păstrează starea jocului și scorurile pentru sesiuni viitoare.

2 Funcționalități ale Jocului

2.1 Sistemul de Mișcare

- Control prin taste(săgeți).
- Viteză dublă de mișcare cu un power-up de viteză.
- Detectarea coliziunii cu pereții.

2.2 Sistemul de Salvare

- Persistența scorurilor utilizând fișiere JSON.
- Menținerea istoricului scorurilor.

3 Interfața Utilizator

3.1 Interfața Principală a Jocului

- **Afișare în timp real a informațiilor:** scorul curent, starea sănătății, power-up-uri active
- **Vizualizarea labirintului.**

3.2 Meniu

- **Joc Nou:** Începe o sesiune nouă, având posibilitatea de a selecta nivelul de dificultate.
- **Pauză** Suspendă temporar jocul, permițând jucătorului să-l reia când dorește.
- **Scoruri Maxime:**: Afișează o listă cu cele mai bune scoruri ale jucătorilor, extrase din fișierul JSON.
- **Iesire:** Oferă opțiunea de a părăsi jocul.

4 Analiza structurii pe baza laboratoarelor

4.1 Laborator 1: Tipuri primitive de date

4.1.1 Utilizarea tipului int

Coordonatele în labirint sunt reprezentate prin variabile de tip int, care definesc pozițiile jucătorului pe axele x și y.

```
public class Player {
    private int row;    // Pozitia pe rand
    private int col;    // Pozitia pe coloana
    private int health; // Sanatatea jucatorului (0 -100)
    private int score;  // Scorul curent
}
```

Aceste variabile sunt esențiale pentru a urmări poziția jucătorului și starea jocului.

4.1.2 Utilizarea tipului String

Numele jucătorilor sunt stocate într-o variabilă de tip String.

```
public class MazeRunnerGUI extends JFrame {
    private String playerName;
    // ...
}
```

4.1.3 Utilizarea tipului boolean

Matricea pereților este implementată folosind un tip boolean, unde false reprezintă un drum liber, iar true un perete.

```
public class Maze {
    private final boolean[][] walls; // false = drum liber, true =
        perete
    // ...
    public boolean isWall(int row, int col) {
        return walls[row][col];
    }
}
```

Aceasta este o alegere eficientă din punct de vedere al memoriei, deoarece un boolean utilizează doar un singur bit.

4.1.4 Utilizarea tipului double

Densitatea pereților care determină dificultatea jocului este reprezentată printr-o valoare de tip double.

```
public enum DifficultyLevel {
    EASY(0.2),    // 20% densitate pereti
    MEDIUM(0.35), // 35% densitate pereti
    HARD(0.5);    // 50% densitate pereti

    private final double wallDensity;
}
```

4.2 Laborator 2: Structuri de control

4.2.1 Structuri de control de selecție (condiționale):

- if: Verificarea dacă o mișcare este validă înainte de a o efectua.

```
private void handleMovement(int newRow, int newCol) {
    if (maze.isValidMove(newRow, newCol)) {
        if (player.hasSpeedBoost()) {
            // Permite miscare dubla cu speed boost
            int deltaRow = newRow - player.getRow();
            int deltaCol = newCol - player.getCol();
            if (maze.isValidMove(newRow + deltaRow, newCol +
                deltaCol)) {
                    newRow += deltaRow;
                    newCol += deltaCol;
            }
        }

        player.move(newRow, newCol);

        if (maze.isExit(newRow, newCol)) {
                handleLevelComplete();
        } else if (maze.isPowerUp(newRow, newCol)) {
                handlePowerUpCollection(newRow, newCol);
        }

        drawMaze();
        }
}
```

- switch: este utilizat pentru a seta nivelul de dificultate al labirintului în funcție de valoarea selectată de utilizator.Este mai lizibil și mai eficient în unele situații decât utilizarea mai multor instrucțiuni if-else.

```
switch (selectedDifficulty) {
    case "Medium":
        difficulty =
            MazeRunner.Modules.Maze.DifficultyLevel.MEDIUM;
        break;
    case "Hard":
        difficulty =
            MazeRunner.Modules.Maze.DifficultyLevel.HARD;
        break;
    default:
        difficulty = Maze.DifficultyLevel.EASY;
        break;
    }
```

4.2.2 Structuri de control de repetare (bucle):

- while: În metoda ensurePathToExit, bucla while este utilizată pentru a garanta crearea unui traseu valid de la poziția de start (startRow, startCol) la poziția de final (endRow, endCol) într-un labirint.

```
    private void ensurePathToExit(int startRow, int startCol, int
        endRow, int endCol) {
    int currentRow = startRow;
    int currentCol = startCol;

    while (currentRow != endRow || currentCol != endCol) {
        if (currentRow < endRow) {
            currentRow++;
        } else if (currentCol < endCol) {
            currentCol++;
        }
        walls[currentRow][currentCol] = false;
    }
}
```

- for: este utilizată pentru a itera prin direcțiile posibile (sus, jos, stânga, dreapta) in interiorul funcţiei dfs.

```
    private void dfs(int row, int col, Random random) {
        int[][] directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
        shuffleArray(directions, random);

         for (int[] dir : directions) {
            int newRow = row + dir[0] * 2;
            int newCol = col + dir[1] * 2;

            if (isInBounds(newRow, newCol) &&
                walls[newRow][newCol]) {
                 walls[row + dir[0]][col + dir[1]] = false;
                 walls[newRow][newCol] = false;
                 dfs(newRow, newCol, random);
            }
        }
    }
```

4.3 Laborator 3: Clase. Mosteniri

4.4 Ierarhia Claselor

Clase de bază

- **Maze**: Generarea și gestionarea labirintului.
- **Player**: Starea și acțiunile jucătorului.
- **PowerUp**: Definește puterile.
- **GameState**: Singleton pentru starea globală a jocului
- **HighScore**; Gestionează scorurile și salvarea acestora in fişier

Clase GUI

- **MazeRunnerGUI extends JFrame**: Interfața principală a jocului
- **StartGameConfig extends JDialog**:Configurarea inițială a jocului.

4.5 Exemplu detaliat de moştenire

```
public class MazeRunnerGUI extends JFrame {
    // Mostenire din JFrame pentru functionalitati de fereastra
    private final Maze maze;
    private final Player player;

    public MazeRunnerGUI(int rows, int cols,
        Maze.DifficultyLevel difficulty) {
         super("Maze Runner"); // Apel constructor parinte
         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         setSize(800, 700);
         setResizable(false);

         this.maze = new Maze(rows, cols, difficulty);
         this.player = new Player();

         setupFrame();
         createMenu();
         setupGameInfoPanel();
         drawMaze();
    }
}
```

În acest exemplu, MazeRunnerGUI moștenește din JFrame, ceea ce îi permite să utilizeze toate funcționalitățile unei ferestre grafice Java.

4.6 Laborator 4: Interfeţe

4.6.1 Interfeţe implementate

- **IGameState**: Definește comportamentele generale ale stării jocului. Aceasta include metode pentru salvarea și încărcarea stării jocului, precum și gestionarea progresului și scorurilor.

```
public interface IGameState {
    void saveHighScore(String playerName, int score);
    void saveCurrentScore();
    void showHighScores();
    void setPlayerName(String playerName);
    void addScore(int points);
}
```

- **IMaze**: Definirea comportamentului pentru gestionarea labirintului, inclusiv generarea și modificarea acestuia.

```
public interface IMaze {
     boolean isValidMove(int row, int col);
     boolean isExit(int row, int col);
     boolean isWall(int row, int col);
     boolean isPowerUp(int row, int col);
     PowerUp getPowerUpAt(int row, int col);
     void removePowerUp(PowerUp powerUp);
     void resetMaze(Maze.DifficultyLevel difficulty);
     int getRows();
     int getCols();
}
```

- **IMazeRunnerGUI**: Interfața pentru clasa care gestionează interfața grafică a jocului. Acesta include metode pentru actualizarea imaginii labirintului, a jucătorului și pentru gestionarea inputului utilizatorului.

```
public interface IMazeRunnerGUI {
void startNewGame();
void togglePause();
void updateGameInfo();
```

```
    // ...
}
```

- **IPlayer**: Interfața care definește comportamentele legate de jucător. Aceasta include metode pentru mutarea jucătorului și gestionarea stării acestuia.

```
public interface IPlayer {
    void move(int newRow, int newCol);
    void resetPlayer();
    int getRow();
    int getCol();
    int getHealth();
    int getScore();
}
```

- **IPowerUp**: Interfața pentru gestionarea power-up-urilor din joc.

```
public interface IPowerUp {
    int getRow();
    int getCol();
    String getType();
    int getDuration();
}
```

4.6.2 Interfeţe grafice

Pentru interfețele grafice, am implementat clase care extind componentele native ale Java Swing pentru a crea o interfață ușor de utilizat și intuitivă:

- **MazeRunnerGUI extends JFrame**: Fereastra principală a aplicației, care gestionează afișarea labirintului și interacțiunea cu utilizatorul. Aici sunt afișate diferitele elemente ale jocului și sunt gestionate evenimentele de input (de exemplu, mișcarea jucătorului, colectarea power-up-urilor).

```
public class MazeRunnerGUI extends JFrame implements
    IMazeRunnerGUI {
    private String playerName;
    private final MazeRunner.Modules.Maze maze;
    private final Player player;
    private final JPanel mazePanel;
    private final JPanel gameInfoPanel;
    private final JLabel timerLabel;
    private final JLabel scoreLabel;
    private final JLabel healthLabel;
    private final JLabel powerUpsLabel;
    private int seconds = 0;
    private Timer gameTimer;
    private final MazeRunner.Modules.GameState gameState;
    private boolean isPaused = false;

    // ...
}
```

- **StartGameConfig extends JDialog**: Această clasă gestionează dialogul pentru configurarea jocului (alegerea nivelului de dificultate, dimensiunea labirintului și numele jucătorului).

```
public class StartGameConfig extends JDialog{
    private String playerName = "";
    private MazeRunner.Modules.Maze.DifficultyLevel
        selectedDifficulty ;
    private boolean startGame = false;
    private JTextField nameField;
    private JComboBox<MazeRunner.Modules.Maze.DifficultyLevel>
        difficultyCombo;
```

```
        public StartGameConfig(Frame parent){
            super(parent, "Maze Runner - New Game", true);
            setupDialog();
        }
        // ...
    }
```

4.7 Laborator 5: Stocarea datelor

Datele de joc, inclusiv scorurile și numele jucătorilor, sunt stocate într-un fișier JSON. Această abordare permite salvarea și încărcarea facilă a scorurilor în cadrul aplicației. Fișierul folosit pentru stocarea scorurilor este denumit **highscores.json** și este gestionat prin clasa **GameState**.

Clasa **GameState** este responsabilă de gestionarea scorurilor și a stării jocului. Aceasta implementează o abordare de tip Singleton, asigurându-se că există o singură instanță a acesteia pe parcursul execuției jocului. Funcțiile principale includ:

- **loadHighScores**: Încarcă scorurile anterioare din fișierul **highscores.json**. Dacă fișierul nu există, returnează o listă goală de scoruri.
- **saveHighScoresToFile**: Salvează scorurile curente într-un fișier JSON, folosind librăria Jackson pentru serializare.
- **addScore**: Adaugă un punctaj nou și îl salvează în fișier.
- **showHighScores**: Afișează scorurile înregistrate, limitând afișarea la top 10 scoruri.

Implementare pentru salvarea și încărcarea scorurilor din fișier JSON:

```
private List<HighScore> loadHighScores() {
    ObjectMapper objectMapper = new ObjectMapper();
    try{
        File file = new File(HIGH_SCORES_FILE);
        if(file.exists()){
            return objectMapper.readValue(file,
                    objectMapper.getTypeFactory().constructCollectionType(List.class,
                        HighScore.class));
        } else {
            return new ArrayList<>();
        }
    } catch (IOException e) {
        logger.severe("Failed to load high scores: " + e.getMessage());
        return new ArrayList<>();
    }
}

private void saveHighScoresToFile() {
    ObjectMapper objectMapper = new ObjectMapper();
    try{
        objectMapper.writerWithDefaultPrettyPrinter().writeValue(new
            File(HIGH_SCORES_FILE), highScores);
    } catch (IOException e) {
        logger.severe("Failed to save high scores: " + e.getMessage());
    }
}
```

Această metodă de stocare este eficientă, asigurându-se că datele sunt persistente între sesiuni de joc, iar fișierele JSON pot fi ușor citite și modificate de către aplicație.

4.8 Laborator 6: JUnit Tests

Pentru a asigura corectitudinea și stabilitatea aplicației, au fost implementate teste unitare utilizând framework-ul JUnit. Aceste teste sunt esențiale pentru validarea funcționalităților aplicației și pentru a asigura că modificările viitoare nu vor introduce erori neprevăzute.

Pentru a rula testele unitare, este necesar să ai JUnit inclus ca dependență în proiectul tău. În cazul utilizării unui manager de tip Gradle, este necesară adăugarea urmatoarei dependenţe :

```
testImplementation("org.junit.jupiter:junit-jupiter:5.10.0")
```

4.8.1 Testarea metodelor ce aparţin interfeţei IPowerUp

Testul pentru metoda getRow():

- **Scop**: Verifică faptul că metoda returnează corect valoarea rândului (în acest caz 5) pentru un obiect **PowerUp**
- Testul presupune crearea unui obiect **PowerUp** cu rândul 5 și coloană 10 și se așteaptă ca valoarea rândului să fie 5.

Testul pentru metoda getRow()getCol():

- **Scop**: Verifică dacă metoda returnează corect valoarea coloanei (în acest caz 10) a unui obiect **PowerUp**.
- Testul creează un obiect **PowerUp** cu rândul 5 și coloana 10 și se așteaptă ca metoda să returneze 10.

Testul pentru metoda getRow()getType():

- **Scop**: Verifică dacă metoda **getType()** returnează corect tipul de power-up (în acest caz, "Speed").
-

Testul pentru metoda getRow()getDuration():

- **Scop**: Verifică corectitudinea valorii pentru durata power-up-ului.
- Testul creează un obiect **PowerUp** cu durata 15 și se așteaptă ca metoda să returneze valoarea 15.

```
package MazeRunner.Interfaces;

import MazeRunner.Modules.PowerUp;
import org.junit.jupiter.api.Test;

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

class IPowerUpTest {

    @Test
    void getRow() {
        PowerUp powerUp = new PowerUp(5, 10, "Speed", 15);
        assertEquals(5, powerUp.getRow(), "Row should be 5");
    }

    @Test
    void getCol() {
        PowerUp powerUp = new PowerUp(5, 10, "Speed", 15);
        assertEquals(10, powerUp.getCol(), "Column should be 10");
    }

    @Test
    void getType() {
        PowerUp powerUp = new PowerUp(5, 10, "Speed", 15);
        assertEquals("Speed", powerUp.getType(), "Type should be
            'Speed'");
    }

     @Test
    void getDuration() {
```

```
        PowerUp powerUp = new PowerUp(5, 10, "Speed", 15);
        assertEquals(15, powerUp.getDuration(), "Duration should be
            15");
    }
}
```

5 Utilizarea bibliotecii Swing

În cadrul acestui proiect, **biblioteca Swing** este utilizată pentru a construi și gestiona interfața grafică a jocului. Swing oferă un set bogat de componente pentru crearea unui GUI interactiv. Principalele utilizări ale acestei biblioteci sunt următoarele:

1. **Fereastra principală a jocului:**
   - JFrame este utilizat pentru a crea fereastra principală a jocului. Aceasta include labirintul generat, scorul și butoanele de control.
2. **Panouri pentru organizarea elementelor**

   Sunt utilizate panouri JPanel pentru a structura diferite secțiuni din interfață:
   - mazePanel: Panou pentru desenarea labirintului.
   - gameInfoPanel: Panou pentru afișarea informațiilor legate de joc (scor, timp, sănătate, power-ups).

   ```
   mazePanel.setLayout(new GridLayout(maze.getRows(), maze.getCols(),
       1, 1));
   gameInfoPanel.setLayout(new GridLayout(1, 4, 10, 0));
   ```
3. **Componente de afișare: JLabel**
   - Sunt utilizate pentru afișarea informațiilor statice sau dinamice, cum ar fi timpul, scorul, sănătatea și power-ups.
4. **Meniu pentru funcționalități: JMenuBar**
   - Un meniu principal este creat pentru gestionarea jocului, incluzând opțiuni precum New Game, Pause, High Scores, și Exit.

     ```
     JMenu gameMenu = new JMenu("Game");
     JMenuItem newGame = new JMenuItem("New Game");
     JMenuItem pause = new JMenuItem("Pause");
     ```
   - Acțiuni asociate: addActionListener() pentru a lega funcții de interacțiunile utilizatorului
5. **Dialoguri și mesaje: JOptionPane**

   Utilizate pentru afișarea de mesaje și interacțiuni rapide, cum ar fi pauza jocului, completarea unui nivel sau selectarea dificultății.

   ```
   JOptionPane.showMessageDialog(this, "Game Paused\nChoose an action:",
       "Paused", JOptionPane.INFORMATION_MESSAGE);
   ```

6 Arhitectura aplicaţiei - UML

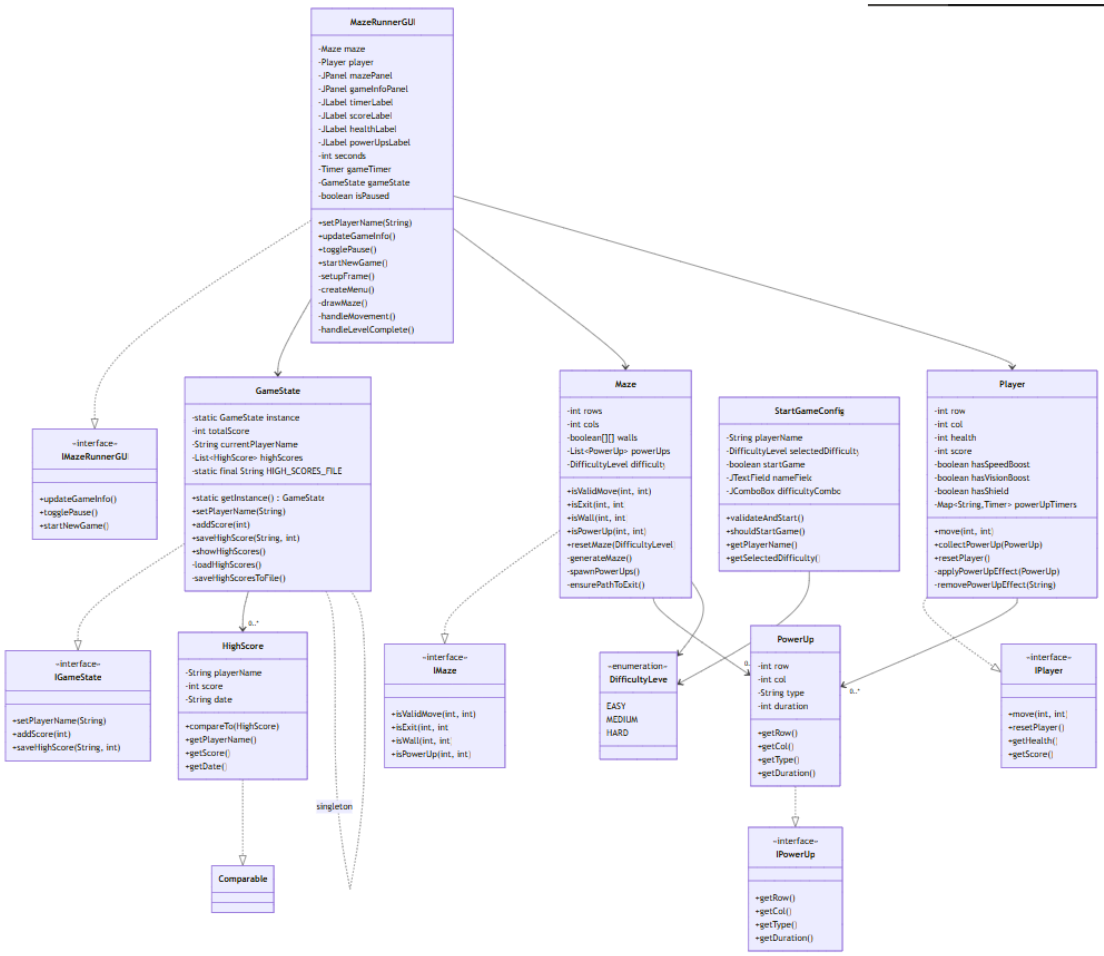

Figura 1: Grafic - UML

7 Îmbunătățiri pentru aplicaţie

- Niveluri multiple de labirint.
- Inamici sau obstacole.
- Efecte sonore și muzică de fundal.
- Clasament online.
- Personaje multiple pentru jucători.

Cuprins

1 Introducere şi obiective **1**
- 1.1 Descriere generală . . . 1
- 1.2 Obiective Principale . . . 1

2 Funcționalități ale Jocului **1**
- 2.1 Sistemul de Mișcare . . . 1
- 2.2 Sistemul de Salvare . . . 1

3 Interfața Utilizator **2**
- 3.1 Interfața Principală a Jocului . . . 2
- 3.2 Meniu . . . 2

4 Analiza structurii pe baza laboratoarelor **2**
- 4.1 Laborator 1: Tipuri primitive de date . . . 2
  - 4.1.1 Utilizarea tipului int . . . 2
  - 4.1.2 Utilizarea tipului String . . . 2
  - 4.1.3 Utilizarea tipului boolean . . . 2
  - 4.1.4 Utilizarea tipului double . . . 3
- 4.2 Laborator 2: Structuri de control . . . 3
  - 4.2.1 Structuri de control de selecție (condiționale): . . . 3
  - 4.2.2 Structuri de control de repetare (bucle): . . . 4
- 4.3 Laborator 3: Clase. Mosteniri . . . 4
- 4.4 Ierarhia Claselor . . . 4
- 4.5 Exemplu detaliat de moştenire . . . 5
- 4.6 Laborator 4: Interfeţe . . . 5
  - 4.6.1 Interfeţe implementate . . . 5
  - 4.6.2 Interfeţe grafice . . . 6
- 4.7 Laborator 5: Stocarea datelor . . . 7
- 4.8 Laborator 6: JUnit Tests . . . 7
  - 4.8.1 Testarea metodelor ce aparţin interfeţei **IPowerUp** . . . 8

5 Utilizarea bibliotecii Swing **9**

6 Arhitectura aplicaţiei - UML **10**

7 Îmbunătățiri pentru aplicaţie **10**