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** și implementarea unei **interfețe grafice** intuitive.

Labirintul este generat procedural folosind un algoritm **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:
  1. **ușor** - 20% pereți
  2. **ușor** - 35% pereți
  3. **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 Survey și Analiză Comparativă

2.1 Analiza Jocurilor Similare

Pac-Man (1980) reprezintă un standard în industrie pentru jocurile bazate pe labirint. Atât Pac-Man, cât și MazeRunner au în comun navigarea prin labirint și sistemul de power-up-uri. Diferența principală între cele două jocuri constă în generarea procedurală a labirinturilor din MazeRunner, în comparație cu labirintul static din Pac-Man.

The Maze Runner (2014) implementează conceptul de labirint într-un mediu 3D cu grafică avansată. Sistemele de gameplay din acest joc includ elemente complexe de explorare și narațiune. Aplicaţia, pe de altă parte, adoptă o abordare 2D, concentrându-se pe mecanici de bază și generare

procedurală.

Modificările aduse de **Minecraft** pentru labirinturi demonstrează eficiența generării procedurale în crearea de conținut variabil. Aceste moduri operează într-un mediu 3D și permit interacțiuni complexe cu mediul, în timp ce MazeRunner implementează concepte similare, dar în 2D.

2.2 Analiza Tehnologiilor Alternative

2.2.1 Platforme de dezvoltare

Unity oferă capacități atât pentru modele 3D, cât și pentru 2D, și suportă dezvoltarea multi-platformă. Unity vine cu instrumente de dezvoltare integrate, dar necesită o licență pentru funcționalități avansate și cerințe de sistem de minim 8GB RAM și un procesor modern.

Java Swing, pe de altă parte, suportă aplicații 2D și are portabilitate pe orice sistem cu JRE. De asemenea, are cerințe de sistem minime, de doar 2GB RAM, și beneficiază de o licență open-source. În plus, Swing se integrează direct cu bibliotecile Java standard, ceea ce îl face o alegere eficientă pentru aplicațiile 2D.

LibGDX este un framework specializat pentru jocuri, care oferă suport cross-platform și optimizări pentru randarea 2D. Acesta are o documentație tehnică limitată, dar beneficiază de o comunitate de dezvoltare activă.

2.3 Alegerea Algoritmului de Generare

Depth-First Search (DFS) are o complexitate temporală de **O(n)** și o complexitate spațială de **O(n)**. Acesta garantează conectivitate completă și generează căi cu lungime maximă. Implementarea sa este bazată pe stivă, ceea ce îl face eficient pentru generarea labirinturilor.[3]

Algoritmul lui **Prim**, cu o complexitate temporală de **O(n log n)** și o complexitate spațială de **O(n)**, generează ramificații multiple și necesită structuri de date adiționale. Acesta folosește pattern-uri de generare neuniforme, oferind o diversitate în structura labirintului.

Recursive Division[3] are o complexitate temporală de **O(n log n)** și o complexitate spațială de **O(log n)**. Algoritmul generează pattern-uri geometrice regulate și are o implementare recursivă. Acesta permite un control precis asupra structurii labirintului generat.

3 Funcționalități ale Jocului

3.1 Sistemul de Mișcare

- Control prin taste(săgeți).
- Power-up-uri și efecte
  - Viteză dublă de mișcare (durată: 10 - 30 secunde)
  - Vedere extinsă (durată: 10 - 30 secunde)
  - Imunitate la capcane (durată: 10 - 30 secunde)
- Mecanici de joc
  - Detectarea coliziunii cu pereții
  - Sistem de viață

3.2 Sistemul de Salvare

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

4 Structura aplicaţiei

4.1 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**: Fereastra principală a aplicației, care gestionează afişarea 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();
            }
            // ...
        }
```

4.2 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();
    }
```

4.3 Arhitectura aplicaţiei - UML

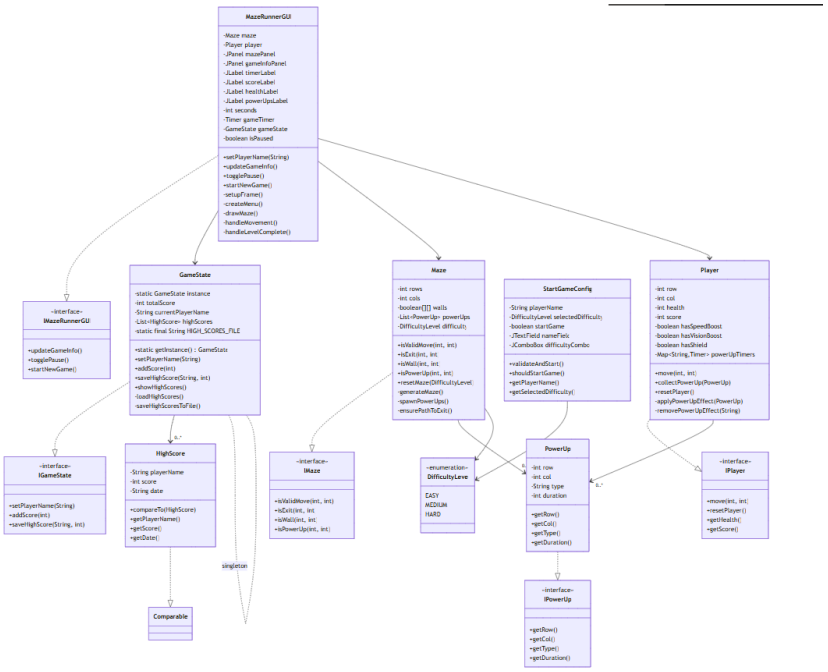

Figura 1: Diagrama - UML

5 Tehnologii utilizate

5.1 Java SE (Standard Edition)

Java SE (Java Platform, Standard Edition) este o platformă de dezvoltare pentru aplicații Java, furnizată de **Oracle**, care oferă un set complet de API-uri și instrumente necesare pentru dezvoltarea și rularea aplicațiilor Java pe desktop, servere și alte medii. Java SE include mașina virtuală Java (JVM), biblioteca de clase de bază, compilatorul și diverse instrumente pentru dezvoltare.

Caracteristici principale ale Java SE:

1. **JVM (Java Virtual Machine)** - Permite rularea codului Java independent de platformă prin intermediul interpretării bytecode-ului.
2. **JDK (Java Development Kit)** – Un pachet de dezvoltare care conține:
   - **Compilatorul javac** pentru traducerea codului sursă Java în bytecode.
   - **JRE (Java Runtime Environment)**, care include JVM și bibliotecile necesare pentru a rula aplicații Java.
   - Instrumente de depanare și analiză (de exemplu, **jconsole, jdb, jvisualvm**).
3. **Biblioteci de bază (Core Libraries)** - Un set de API-uri standard pentru manipularea colecțiilor (**java.util**), gestionarea fișierelor (**java.io, java.nio**), rețelei (**java.net**), multi-threading (**java.util.concurrent**), bazelor de date (**JDBC**), securității și criptografiei (**java.security**).
4. **JavaFX și Swing/AWT** – API-uri pentru dezvoltarea interfețelor grafice de utilizator (GUI).
5. **Java Logging API (java.util.logging)** – Pentru înregistrarea și gestionarea mesajelor de log.
6. **Java Logging API (java.util.logging)** – API pentru conectarea și interacțiunea cu baze de date.

Utilizare în proiect:

- **Colecții:**

```
// In GameState.java
private List<HighScore> highScores;
// In Player.java
private final Map<String, Timer> powerUpTimers;
```

- **Gestionarea fișierelor:**

```
// In GameState.java
File file = new File(HIGH_SCORES_FILE);
if(file.exists()) {
    // procesare fisier
}
```

- **Data și timp:**

```
// In HighScore.java
this.date = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new
    Date());
```

5.2 Java Swing (GUI)

Swing este o bibliotecă Java pentru dezvoltarea **interfețelor grafice (GUI - Graphical User Interface)** în aplicațiile desktop. Aceasta face parte din pachetul **Java Foundation Classes (JFC)** și oferă un set de componente GUI, cum ar fi feronerie (buton, etichete, câmpuri de text, meniuri, etc.), controale de dialog și multe altele. Este un framework de nivel înalt, bazat pe un model de evenimente, care permite crearea de aplicații grafice într-un mod flexibil și personalizabil.

În cadrul acestui proiect, **biblioteca Swing** este utilizată pentru a construi și gestiona interfața grafică a jocului.

Componente principale utilizate:

- **JFrame** - Fereastra principală

```
// In MazeRunnerGUI.java
public class MazeRunnerGUI extends JFrame implements IMazeRunnerGUI {
    private final JPanel mazePanel;
    private final JLabel timerLabel;
    private final JLabel scoreLabel;
}
```

- **Layout Managers:**

```
// In StartGameConfig.java
JPanel configPanel = new JPanel(new GridBagLayout());
// In MazeRunnerGUI.java
setLayout(new BorderLayout(10, 10));
```

- **Componente UI:**

```
// In MazeRunnerGUI.java
// Meniuri
JMenuBar menuBar = new JMenuBar();
JMenu gameMenu = new JMenu("Game");
JMenuItem newGame = new JMenuItem("New Game");

// Labels si Panouri
JLabel cell = new JLabel();
cell.setHorizontalAlignment(SwingConstants.CENTER);
cell.setOpaque(true);
```

- **Event Handling:**

```
// In MazeRunnerGUI.java
private class MazeKeyListener implements KeyListener {
    @Override
    public void keyPressed(KeyEvent e) {
        switch (e.getKeyCode()) {
            case KeyEvent.VK_UP:
                newRow--;
                break;
            // ...
        }
    }
}
```

- **Timers:**

```
// In MazeRunnerGUI.java
gameTimer = new Timer(1000, e -> {
    if (!isPaused) {
        seconds++;
        timerLabel.setText("Time: " + seconds + "s");
    }
});
```

5.3 Jackson (JSON Processing)

JSON (JavaScript Object Notation)[6] este un format de reprezentare a datelor structurate, utilizat pentru schimbul de informații între sisteme.

Acesta este un format text simplu, bazat pe standardul JavaScript, care folosește un set limitat de reguli pentru a descrie obiecte și structuri de date, utilizând perechi cheie-valoare și secvențe ordonate de valori. JSON este independent de limbajul de programare, fiind ușor de generat și interpretat de diverse platforme și aplicații software.

Structurile fundamentale ale JSON sunt:

1. **Obiecte:** Colecții de perechi cheie-valoare, reprezentate între acolade { }.
2. **Arrays (liste):** Colecții ordonate de valori, reprezentate între paranteze drepte [ ].
3. **Valori:** Pot fi de tipul: șiruri de caractere (string), numere, valori booleene (true/false), obiecte, liste sau null.

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**.

- **Serializare** (Salvare în JSON): Salvează scorurile curente într-un fișier JSON, folosind librăria Jackson pentru serializare.

```
// In GameState.java
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());
    }
}
```

- **Deserializare** (Încărcare din JSON): Încarcă scorurile anterioare din fișierul **highscores.json**. Dacă fișierul nu există, returnează o listă goală de scoruri.

```
// In GameState.java
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));
        }
    } catch (IOException e) {
        logger.severe("Failed to load high scores: " + e.getMessage());
    }
    return new ArrayList<>();
}
```

- **Format JSON rezultat:**

```
[
  {
    "playerName": "Player1",
    "score": 1000,
    "date": "2025-01-15 14:30:00"
  },
  {
    "playerName": "Player2",
    "score": 850,
    "date": "2025-01-15 15:45:00"
  }
]
```

5.4 Java Logging API

Java Logging API este un set de clase și interfețe în cadrul **Java Standard Library** care permite aplicațiilor Java să înregistreze mesaje de jurnalizare (log) într-un mod sistematic și flexibil.

Acesta oferă un mecanism pentru a captura informații de diagnosticare, erori și evenimente importante care apar în timpul execuției unei aplicații, ajutând dezvoltatorii să monitorizeze și să depaneze aplicațiile

Java Logging API face parte din pachetul java.util.logging și include următoarele componente cheie:

- **Logger:** O clasă care permite înregistrarea mesajelor. Este folosit pentru a scrie mesaje de log, care pot fi de diverse tipuri (informative, de avertizare, erori, etc.).
- **Handler:** O clasă responsabilă cu livrarea mesajelor de log într-un anumit loc (exemplu: fișier, consolă, etc.).
- **Formatter:** Permite formatarea mesajelor de log pentru a le face mai ușor de citit sau mai structurate (de exemplu, sub forma unui șir de caractere în format JSON sau XML).
- **LogRecord:** O clasă care reprezintă un mesaj de log cu informații despre data, nivelul de severitate, sursa și mesajul propriu-zis.
- **Level:** O enumerație care definește nivelurile de severitate ale mesajelor de log, cum ar fi **SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST**.

Configurare și utilizare:

```java
// In GameState.java
private static final Logger logger =
    Logger.getLogger(GameState.class.getName());

// Exemple de logging
logger.severe("Failed to save high scores: " + e.getMessage());
logger.warning("Could not load previous high scores");
logger.info("Game started with player: " + playerName);
```

6 Testare și Performanță - Unit Testing

JUnit[7] este un framework open-source pentru testarea unitară în Java, utilizat pentru a valida corectitudinea și robustețea codului prin execuția automată a unor cazuri de test predefinite. Bazat pe principiile **Test-Driven Development (TDD)** și **Extreme Programming (XP)**, JUnit oferă o metodologie formalizată pentru verificarea comportamentului funcțiilor individuale (unități de cod) și facilitează detecția timpurie a defectelor software.

Fiind parte a ecosistemului **xUnit**, JUnit adoptă un model bazat pe adnotări, asercții, fixture-uri de testare și rule engines, permițând dezvoltatorilor să definească și să ruleze teste într-un mod modular și reproductibil. Framework-ul suportă gestionarea ciclului de viață al testelor prin adnotări precum **@BeforeEach, @AfterEach, @BeforeAll și @AfterAll**, iar prin mecanisme precum **assertEquals(), assertTrue() și assertThrows()**, dezvoltatorii pot verifica output-ul codului în raport cu așteptările.

JUnit este compatibil cu **Continuous Integration (CI/CD)** și se integrează cu instrumente populare precum **Maven, Gradle, Jenkins** și **SonarQube**, permițând astfel monitorizarea calității codului și prevenirea regresiilor.

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")
```

În cazul utilizării unui manager de tip **Maven**, este necesară adăugarea dependenţei :

```
<dependencies>
    <!-- JUnit 5 API si runtime -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.10.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

6.1 Testarea metodelor ce aparțin interfeței 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 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 getType():

- **Scop**: Verifică dacă metoda **getType()** returnează corect tipul de power-up (în acest caz, "Speed").
- Testul creează un obiect **PowerUp** cu tipul "Speed" și verifică dacă metoda returnează corect această valoare.

```
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");
    }
}
```

Testul pentru metoda 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.

7 Controale și Interfață

7.1 Navigarea în joc

Jucătorul poate controla personajul folosind tastatura. Controalele sunt următoarele:

- ↑ **(Săgeată sus)** – Muta jucătorul în sus

- ↓ **(Săgeată jos)** – Muta jucătorul în jos
- ← **(Săgeată stânga)** – Muta jucătorul în stânga
- → **(Săgeată dreapta)** – Muta jucătorul în dreapta

7.2 Elemente ale interfeței grafice

Interfața grafică a jocului este împărțită în mai multe secțiuni:

7.2.1 Fereastra principală

Fereastra principală a jocului este un JFrame care conține:

- **Labirintul** – Afișat într-un panou central (JPanel), fiecare celulă fiind reprezentată grafic.
- **Panoul de informații** – Afișează timpul, scorul și starea jucătorului.

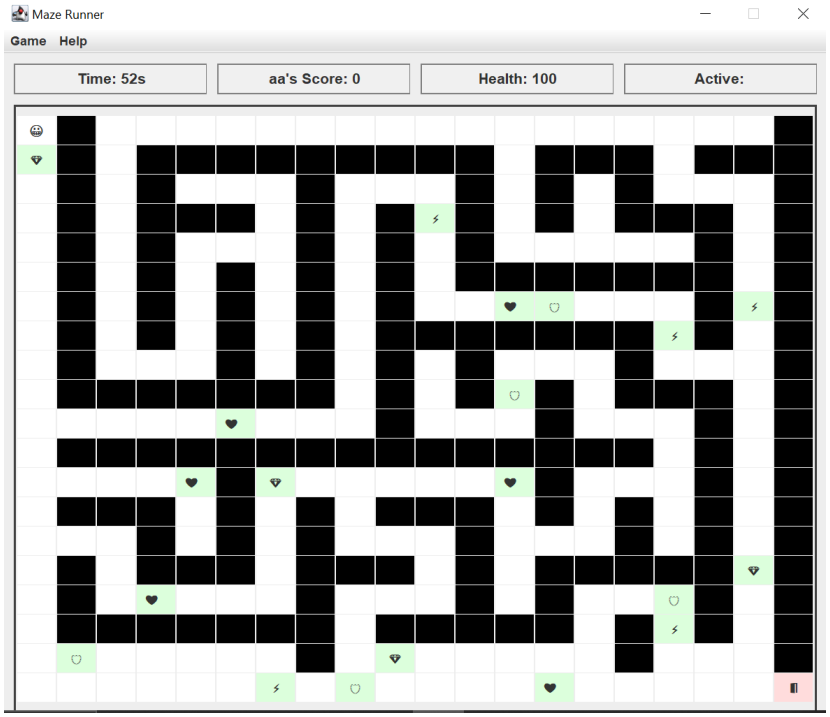

Figura 2: Fereastra principală

7.2.2 Meniul jocului

În partea superioară a ferestrei există o bară de meniu cu următoarele opțiuni:

- **Game**
  - **New Game**: Începe un joc nou, pentru care se deschide o fereastră pop-up pentru alegerea nivelului de dificultate
  - **Pause**: Pune jocul pe pauză
  - **High Scores**: Afișează tabela de scoruri într-o fereastra pop-up
  - **Exit**: Închide aplicația.
- **Help**
  - **How to Play**: Afișează instrucțiuni despre mecanicile jocului
  - **About**: Oferă informații despre joc și dezvoltatori

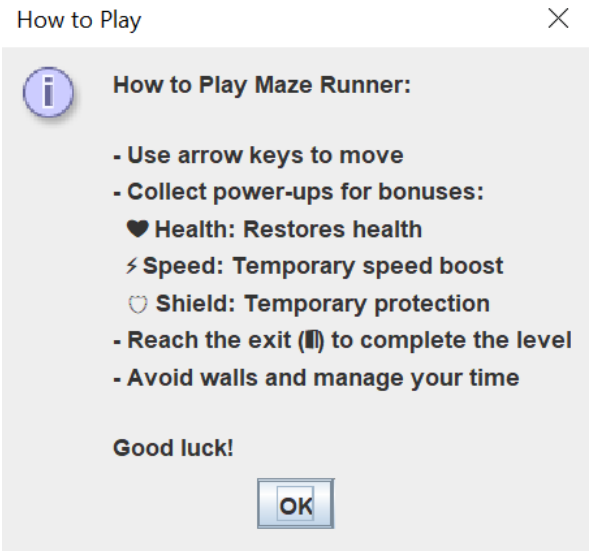

Figura 3: Meniu - How to play

7.2.3 Indicatori vizuali în labirint

Jocul utilizează următorii indicatori vizuali pentru a reprezenta elementele din labirint:

- **(Jucător)** – Poziția actuală a jucătorului în labirint
- **(Ieșire)** – Obiectivul final al jucătorului
- **(Ziduri)** – Pereți ai labirintului care nu pot fi traversați
- **(Power-ups)** – Elemente speciale care pot fi colectate

7.2.4 Fereastra de configurare a aplicaţiei

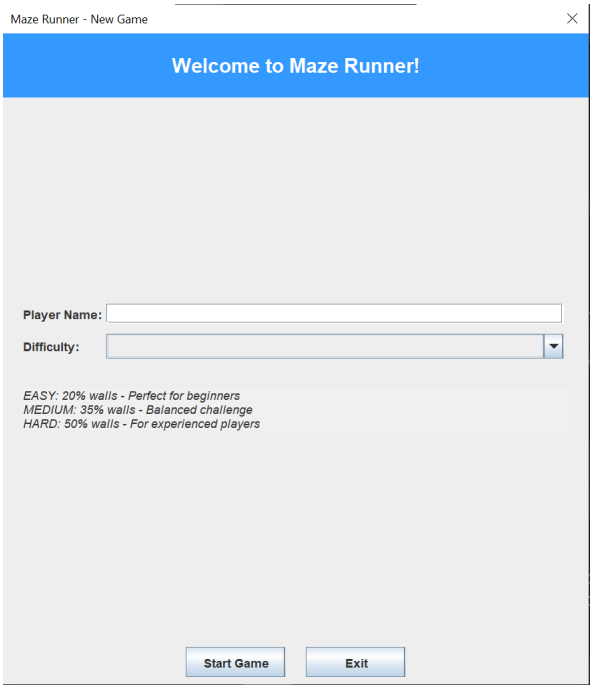

Figura 4: Fereastra de configurare a jocului

8 Cerinţe de sistem

Hardware minim:

- Procesor: 2GHz
- RAM: 4GB
- Spațiu disk: 100MB

Software necesar:

- Java Runtime Environment 8 sau superior
- Sistem de operare: Windows/Linux/MacOS

9 Contributie personală

9.1 Implementări Originale

9.1.1 Algoritmul de Generare a Labirintului

Am implementat algoritmul DFS (Depth-First Search) pentru generarea procedurală a labirinturilor, asigurându-mă că acesta poate susține trei niveluri de dificultate. Algoritmul a fost optimizat pentru a genera rapid labirinturi de dimensiuni mari, având în vedere performanța. În plus, am garantat existența unui drum valid între start și ieșire pentru fiecare labirint generat.

9.1.2 Interfața Grafică

Am proiectat și implementat o interfață grafică intuitivă utilizând Java Swing, cu scopul de a oferi o experiență ușor de utilizat. Am creat un sistem de meniuri simplu de navigat, care permite acces rapid la opțiunile jocului, și am implementat feedback vizual pentru acțiunile jucătorului, îmbunătățind astfel interactivitatea. De asemenea, am adăugat elemente de UI care permit afișarea scorului și a timpului jucătorului în timpul jocului.

9.2 Provocări Tehnice

Una dintre provocările tehnice importante a fost generarea rapidă a labirinturilor de dimensiuni mari. Am rezolvat această problemă implementând un algoritm optimizat, care asigură o generare eficientă. O altă provocare a fost sincronizarea timer-elor pentru power-up-uri, iar pentru aceasta am implementat un sistem centralizat de gestionare a efectelor temporare.

9.3 Îmbunătățiri ale Codului

Am refactorizat codul pentru a îmbunătăți modularitatea și ușurința de întreținere, aplicând cele mai bune practici de programare. Am adăugat teste unitare pentru componentele critice ale jocului, asigurându-mă că funcționează corect în diverse condiții.

9.4 Experiență Dobândită

Am aprofundat cunoștințele în Java și programare orientată pe obiecte, consolidându-mi astfel abilitățile tehnice. Am dobândit experiență semnificativă în dezvoltarea interfețelor grafice cu Swing, învățând cum să creez UI-uri eficiente și ușor de utilizat. De asemenea, am învățat să optimizez aplicațiile și să le depurez, pentru a îmbunătăți performanța acestora. În plus, am înțeles importanța design patterns în dezvoltarea software și am aplicat aceste concepte pentru a crea soluții scalabile și ușor de întreținut.

10 Bibliografie

Bibliografie

[1] D. B. Wilson, "Generating Random Spanning Trees More Quickly than the Cover Time," *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, pp. 296–303, 1996.

[2] D. J. Aldous, "The Random Walk Construction of Uniform Spanning Trees and Uniform Labelled Trees," *SIAM Journal on Discrete Mathematics*, vol. 3, no. 4, pp. 450–465, 1990.

[3] J. Buck, "Recursive Division Algorithm," Available at: `https://weblog.jamisbuck.org/2011/1/12/maze-generation-recursive-division-algorithm`, [Accessed: 19-Jan-2025].

[4] H. Schildt, *Java: The Complete Reference, Eleventh Edition*, McGraw Hill, 2021.

[5] D. Eck, "Introduction to Programming Using Java, Version 9," Available at: `http://math.hws.edu/javanotes/`, [Accessed: 19-Jan-2025].

[6] Oracle, "Java JSON Processing API," Available at: `https://docs.oracle.com/en/java/javase/11/docs/api/java.json/module-summary.html`, [Accessed: 19-Jan-2025].

[7] JUnit Team, "JUnit 5 User Guide," Available at: `https://junit.org/junit5/docs/current/user-guide/`, [Accessed: 19-Jan-2025].

Cuprins