

Dokumentacja końcowa projektu

Andrii Rybachok, Oleksandr Chaienko, Zespół 1

06.06.2024

1 Opis projektu

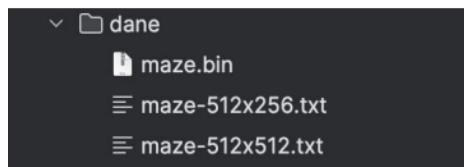
Aplikacja graficzna która pozwala rozwiązywać labirynty i zapisywać ich rozwiązania. Podstawową funkcjonalnością programu jest znalezienie najkrótszej ścieżki w labiryncie i wyświetlanie rozwiązania graficznie. Program może wyczytywać labirynt oraz zapisywać rozwiązanie w formacie tekstowym, binarnym oraz png. Wszystkie błędy mają odpowiedni komunikat, który informuje użytkownika o przyczynie błędu. Przy pomocy interfejsu, użytkownik może ładować labirynt, zarządzać pozycjami początku i końca, zapisywać wyniki działania programu.

2 Pliki

2.1 Drzewo programu

- /code
 - /bin
 - /dane
 - /src
 - Makefile

2.2 Przykładowe dane wejściowe



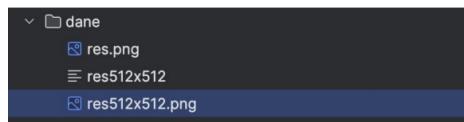
W katalogu dane mamy przykładowe pliki zawierające labirynt w postaci tekstowej albo binarnej. Format tekstowy składa się z czterech symboli:

- X - ściana

- Spacja- ścieżka
- P- początek labiryntu
- K- koniec labiryntu

Format pliku binarnego musi być zgodny z opisem umieszczonym w pliku opis pliku binarnego.pdf.

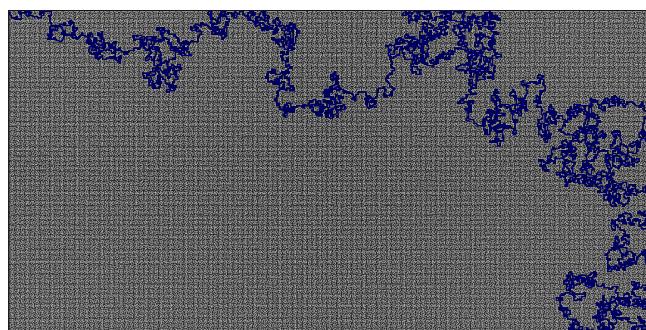
2.3 Przykładowe dane wyjściowe



W tym samym miejscu gdzie są przechowywane dane wejściowe, znajdują się wyniki działania programu na podstawie tych plików. Są przedstawione w rozszerzeniu tekstowym, binarnym oraz obrazka labiryntu w rozszerzeniu (.png). Dla wyniku tekstowego jest wykorzystywany następny format:

```
START
FORWARD 1
TURNLEFT
FORWARD 4
TURNRIGHT
FORWARD 3
STOP
```

Dla pliku binarnego jest wykorzystywany format przedstawiony w pliku opis pliku binarnego.pdf



Przykładowy wygląd rozwiązania png.

3 Moduły

Program jest podzielony na cztery główne moduły:

- Moduł logiki labiryntu (`mazeLogic`)
- Moduł interfejsu graficznego (`gui`)
- Moduł przycisków interfejsu (`gui.buttons`)
- Główny moduł aplikacji (`MainFrame`)

3.1 Główny moduł aplikacji (`MainFrame`)

Jest główną klasą okna aplikacji, która inicjalizuje i zarządza wszystkimi komponentami interfejsu użytkownika.

```
public MainFrame() {  
    setTitle("Maze Solver");  
    setSize( width: 1200, height: 800);  
    setDefaultCloseOperation(EXIT_ON_CLOSE);  
  
    // Główny panel  
    JPanel mainPanel = new JPanel(new BorderLayout());  
  
    // Maze Frame z paskami przewijania  
    mazeFrame = new MazeFrame(new MazeService());  
    JScrollPane scrol = new JScrollPane(mazeFrame);  
    mainPanel.add(scrol, BorderLayout.CENTER);  
}
```

Tworzenie głównego panelu (`mainPanel`) i dodanie do niego paska przewijania (`JScrollPane`), który zawiera `MazeFrame`.

```

// Panel narzędzi
JPanel toolPanel = new JPanel(new FlowLayout());
loadTextButton = new JButton(ButtonTexts.Load);
saveButton = new JButtonButtonTexts.Export();
findPathButton = new JButtonButtonTexts.FindPath();
selectStartButton = new SetStartBtn(mazeFrame, ButtonTexts.SelectStart);
selectEndButton = new SetEndBtn(mazeFrame, ButtonTexts.SelectEnd);
JButton[] buttons = {
    loadTextButton,
    saveButton,
    findPathButton,
    selectStartButton,
    selectEndButton
};
for (JButton jButton : buttons) {
    toolPanel.add(jButton);
    mazeFrame.buttons.add(jButton);
}
mainPanel.add(toolPanel, BorderLayout.NORTH);

// Dodaj główny panel do ramki
add(mainPanel);

// Dodaj funkcje do przycisków
loadTextButton.addActionListener(new LoadButtonTXT(mazeFrame));
saveButton.addActionListener(new SaveButton(mazeFrame));
findPathButton.addActionListener(e -> mazeFrame.solveMaze());
}
}

```

```

public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        @Override
        public void run() { new MainFrame().setVisible(true); }
    });
}
}

```

Tworzenie panelu narzędzi (toolPanel) z przyciskami:

- ładowania labiryntu
- zapisu
- znajdowania ścieżki
- ustawiania punktu startowego i końcowego

3.2 Moduł logiki labiryntu (mazeLogic)

Zawiera klasy odpowiedzialne za logikę labiryntu, takie jak parsowanie labiryntu, reprezentacja grafowa labiryntu oraz usługi związane z rozwiązywaniem labiryntu.

3.2.1 ImazeService

```

// Interfejs obsługujący operacje na labiryntach i reprezentacjach labiryntów
public interface ImazeService {
    // Metoda odczytu listy punktów reprezentujących rozwiązań labiryntu na podstawie danego rozwiązań w postaci listy krawędzi
    public List<Point> getSolutionPoints(MazeParser parser); // Usunięto implementację
    // Metoda odczytu rozwiązań labiryntu i tworząca plik, w którym dla każdego punktu zapisywana jest jego pozycja i krawędź
    public void saveMazeToFile(File file, List<Point> solvePoints, BufferedImage image); // Usunięto implementację
}

```

Interfejs **ImazeService** definiuje metody, które muszą być zaimplementowane przez klasę obsługującą operacje na labiryntach.

3.2.2 MazeConstants

```
public class MazeConstants {
    4 usages
    public static final char Wall = 'X';
    4 usages
    public static final char Path = ' ';
    5 usages
    public static final char Start = 'P';
    3 usages
    public static final char Solution = 'S';
    5 usages
    public static final char End = 'K';
    2 usages
    public static final char TemporaryPoint = 'T';

    2 usages
    public static Color getConstantColor(char symbol) {
        switch (symbol) {
            case MazeConstants.Start:
                return Color.GREEN; // Początkowa pozycja
            case MazeConstants.End:
                return Color.RED; // Koncowa pozycja
            case MazeConstants.Wall:
                return Color.BLACK; // Ściana
            case MazeConstants.Solution:
                return Color.BLUE; // Rozwiązańie
            case MazeConstants.TemporaryPoint:
                return Color.CYAN; // Tymczasowy Klikniety punkt
            case MazeConstants.Path:
                return Color.WHITE; // Tymczasowy Klikniety punkt
            default:
                return Color.ORANGE; // Error
        }
    }
}
```

Ułatwia zarządzanie kolorami wizualizacji labiryntu poprzez zdefiniowanie kolorów dla różnych symboli i zapewnienie prostego sposobu uzyskiwania koloru dla określonego symbolu.

3.2.3 MazeParser

```
// Konstruktor inicjalizujący parser z nazwą pliku labiryntu
class MazeParser {
    public MazeParser(File file) throws IOException {
        var fileParts = file.getName().split("\\.");
        if (fileParts.length > 1) {
            switch (fileParts[1]) {
                case "txt":
                    List<String> lines = new ArrayList<String>(); // lista do przechowywania każdej linii labiryntu
                    try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
                        String line;
                        // Dóczytał każdą linię z pliku i dodaje ją do listy
                        while ((line = reader.readLine()) != null) {
                            lines.add(line);
                        }
                    }
                    maze = lines.substring(0, 20); // Wymuszą listę na 20 tablic
                    findStartAndEnd(); // Znajduje wszystkie startowe i końcowe w labiryntach
                    rows = maze.length;
                    cols = maze[0].length;
                    break;
                case "bin":
                    try (FileInputStream fis = new FileInputStream(file)) {
                    }
                    break;
                default:
                    throw new IOException("Not supported extension. Available extensions: .txt, .bin");
            }
        } else {
        }
    }
}
```

Inicjalizuje parser za pomocą nazwy pliku zawierającego labirynt. Odczytuje każdą linię z pliku i dodaje ją do listy `lines`. Konwertuje listę do dwuwymiarowej tablicy `maze`. Wywołuje metodę `findStartAndEnd()`, aby zlokalizować punkty startu i końca w labiryncie.

```

// Metoda do znajdowania i przechowywania sąsiadów punktów startowego ('S') i końcowego ('E')
private void findStartAndEnd() { // usage
    for (int i = 0; i < maze.length; i++) {
        for (int j = 0; j < maze[i].length; j++) {
            if (maze[i][j] == MazeConstants.Start) {
                startX = i;
                startY = j;
            } else if (maze[i][j] == MazeConstants.End) {
                endX = i;
                endY = j;
            }
        }
    }
}

// Metoda do resetowania punktu startowego
public void resetStart(Point newStart) { // usage
    resetPoint(getStart(), newStart, MazeConstants.Start);
    startX = newStart.x;
    startY = newStart.y;
}

// Metoda do resetowania punktu końcowego
public void resetEnd(Point newEnd) { // usage
    resetPoint(getEnd(), newEnd, MazeConstants.End);
    endX = newEnd.x;
    endY = newEnd.y;
}

// Metoda pomocnicza do resetowania punktu
private void resetPoint(Point oldPos, Point newPos, char symbol) { // usage
    maze[(int) oldPos.getX()][(int) oldPos.getY()] = MazeConstants.Wall;
    maze[(int) newPos.getX()][(int) newPos.getY()] = symbol;
}

```

Przechodzi przez całą tablicę `maze` i znajduje pozycje punktów startu (S) i końca (E). Przypisuje wartości do pól `startX`, `startY`, `endX` i `endY`. Metody `resetStart/End` ustawiają nowy punkt startu i końca w labiryncie.

3.2.4 MazeGraph

```

public class MazeGraph { // usage
    private final char[][] maze; // 2D tablica przechowująca układ labiryntu // usage
    private final int rows; // ilość wierszy labiryntu // usage
    private final int cols; // ilość kolumn labiryntu // usage
    private final Map<Point, List<Point>> graph; // Mapa punktów grafu z listą sąsiadów // usage

    // konstruktor inicjalizujący graf na podstawie układu labiryntu
    public MazeGraph(char[][] maze) { // usage
        this.maze = maze;
        this.rows = maze.length;
        this.cols = maze[0].length;
        buildGraph(); // buduję graf na podstawie układu labiryntu
    }

    // metoda budująca graf na podstawie labiryntu
    private void buildGraph() {
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                for (int k = 0; k < 4; k++) { // sprawdzanie sąsiadów
                    if (maze[i][j] != MazeConstants.Wall) { // nieprzewodzący tylko kiedyś nie bedzie ścianami
                        Point neighbor = new Point(i + dx[k], j + dy[k]); // nowy sąsiad
                        graph.putIfAbsent(i * cols + j, new ArrayList<>()); // inicjalizacja listy sąsiadów
                        for (Point neighbor : getNeighbors(i, j)) {
                            if (maze[neighbor.x][neighbor.y] != MazeConstants.Wall) { // sąsiad nie jest ścianą
                                graph.get(i * cols + j).add(neighbor); // dodaj sąsiada jeśli nie są ścianą
                            }
                        }
                    }
                }
            }
        }
    }
}

```

`MazeGraph` przypisuje przekazaną tablicę do pola `maze`, ustawia liczbę wierszy i kolumn, a następnie buduje graf, wywołując metodę `buildGraph`. `buildGraph` dla każdej komórki, która nie jest ścianą (`MazeConstants.Wall`), tworzy nowy punkt `Point(i, j)` i dodaje go do grafu.

```

// Metoda do pobierania sąsiadów (góra, dół, lewo, prawo) danej komórki
private List<Point> getNeighbors(int x, int y) { 1 usage
    List<Point> neighbors = new ArrayList<>();
    if (x > 0)
        neighbors.add(new Point(x - 1, y)); // Góra
    if (x < rows - 1)
        neighbors.add(new Point(x + 1, y)); // Dół
    if (y > 0)
        neighbors.add(new Point(x, y - 1)); // Lewo
    if (y < cols - 1)
        neighbors.add(new Point(x, y + 1)); // Prawo
    return neighbors;
}

// Metoda zwracająca graf
public Map<Point, List<Point>> getGraph() { return graph; }
}

```

Sprawdza cztery możliwe kierunki (góra, dół, lewo, prawo) i dodaje te, które znajdują się w granicach labiryntu. Tworzy listę punktów Point reprezentujących te sąsiadujące komórki. Metoda `getGraph` zwraca graf w formie mapy.

3.2.5 Dijkstra

```

// metoda realizująca algorytm za pomocą algorytmu Dijkstra
public List<Point> solve(Point start, Point end) {
    Map<Point, Point> prev = new HashMap<>(); // Mapa do przechowywania poprzedniego punktu na ościeżce
    Map<Point, Integer> distances = new HashMap<>(); // Mapa do przechowywania odległości od punktu startowego
    PriorityQueue<Point> pq = new PriorityQueue<Point>(Comparator.comparingInt(distances::get)); // kolejka priorytetowa
                                                                // do algorytmu Dijkstra

    // Inicjalizacja odległości jako nieskończoność
    for (Point node : graph.keySet()) {
        distances.put(node, Integer.MAX_VALUE);
    }
    distances.put(start, 0); // odległość od startu do startu to 0
    pq.add(start); // dodaj punkt startowy do kolejki priorytetowej

    // Pętla do przeszukiwania grafu
    while (pq.isEmpty()) {
        Point current = pq.poll(); // Pobierz punkt z najmniejszą odlegością
        if (current.equals(end)) {
            return reconstructPath(prev); // Instrukcja skończenia, ostatecznie ja
        }

        // Przeszukaj sąsiadów bieżącego punktu
        for (Point neighbor : graph.get(current)) {
            int newDist = distances.get(neighbor) + 1; // Zaktualizuj tempość bieżącego sąsiada
            if (newDist < distances.get(neighbor)) {
                distances.put(neighbor, newDist); // Aktualizuj odległość
                prev.put(neighbor, current); // Aktualizuj poprzedni punkt
                pq.add(neighbor); // Dodaj sąsiada do kolejki priorytetowej
            }
        }
    }
    return Collections.emptyList(); // Nie znaleziono ścieżki
}

```

Metoda `solve()` wykonuje algorytm Dijkstry na podstawie przekazanego grafu, punktu startowego i punktu końcowego. Algorytm Dijkstry jest wykorzystywany do znalezienia najkrótszej ścieżki w grafie, co jest kluczowym elementem w rozwiązywaniu labiryntów. Dzięki temu algorytmowi możliwe jest odnalezienie optymalnej drogi od punktu startowego do punktu końcowego.

```

// Metoda do odtwarzania ścieżki od końca do początku
private List<Point> reconstructPath(Map<Point, Point> prev) { 1 usage
    List<Point> path = new ArrayList<>();
    for (Point st = end; st != null; st = prev.get(st)) {
        path.add(st); // Dodaj każdy punkt na ścieżce do listy
    }
    Collections.reverse(path); // Odwróć listę, aby uzyskać ścieżkę od początku do końca
    return path;
}

```

Metoda `reconstructPath` odtwarza optymalną ścieżkę od punktu końcowego do punktu startowego na podstawie mapy poprzednich punktów.

3.2.6 MazeService

```
public class MazeService implements IMazeService { ... }

    // Metoda do odczytu punktów reprezentujących labirynt
    private void loadPoints() {
        public List<Point> getSavePoints(MazeParser parser) {
            MazeGraph graph = new MazeGraph(parser.getMaze()); // Tworzenie grafu z labiryntem
            Dijkstra dijkstra = new Dijkstra(graph.getGraph(), parser.getStart(), parser.getGoal()); // Inicjalizacja algorytmu Dijkstry
            return dijkstra.getRoute(); // Zwracanie ścieżki
        }
    }
```

Implementuje interfejs **IMazeService** i dostarcza konkretną implementację metod do rozwiązywania labiryntów i zapisywania ich do plików.

```
    // Metoda do odczytu danych binarnych
    public void readBinary() {}

    // Metoda do zapisywania labiryntu do pliku
    public void saveMaze(File file, List<Point> solvePoints, BufferedImage image) {}

    // Metoda do zapisu danych binarnych (pusta)
    private void saveAsBinary(File file, List<Point> solvePoints) {}

    // Metoda do zapisywania ścieżki rozwiązania do pliku testowego
    private void saveSolveStepsToFile(File file, List<Point> solvePoints) {}

    // Metoda opisująca skok posiadana przez użytkownika w labiryncie
    private String describeAction(int index, Point current, List<Point> steps) {}

    // Metoda opisująca kierunek ruchu w labiryncie
    private String describeDirection(int steps, String positive, String negative) {}

    // Metoda opisująca sekwencję ruchów w labiryncie
    private String describeSequence(int count, String action) { return count + ' ' + action; }

    // Metoda do rozwiązywania labiryntu jako PNG
    private void saveMazeAsPng(File file, BufferedImage image) {}

    // Metoda do uzyskania rozszerzenia pliku
    private String getFileExtension(File file) {}
```

Ta klasa odpowiada za główną logikę aplikacji związaną z operacjami na labiryncie, takimi jak rozwiązywanie, zapisywanie i odczytywanie. Ta klasa wykorzystuje algorytmy i mechanizmy przetwarzania danych, aby zapewnić funkcjonalność aplikacji.

3.3 Moduł interfejsu graficznego (gui)

Zapewnia interaktywny interfejs użytkownika do obsługi i wizualizacji labiryntów oraz działań z nimi związanych.

3.3.1 MazeFrame

```
    // Konstruktor klasy MazeFrame
    public MazeFrame(IMazeService mazeService) { ... }

    this.buttons = new ArrayList<Button>();
    this.mazeService = mazeService;
    setBackground(Color.WHITE);
    setPreferredSize(new Dimension(1000, 1000));
    setLayout(null);
    setUndecorated(true);

    // DEBUG: klinicznie myśl
    public void mouseClicked(MouseEvent e) {
        if (parser != null && !parser.isSolved()) {
            int col = e.getX() / MazeConstants.CELL_SIZE;
            int row = e.getY() / MazeConstants.CELL_SIZE;
            var maze = parser.getMaze();

            if (clickedPoint != null) {
                if ((col == clickedPoint.getCol()) && (row == clickedPoint.getRow())) {
                    return;
                }
                repaintOnePixel(clickedPoint, maze[(int) clickedPoint.getCol()][(int) clickedPoint.getRow()]);
            }

            clickedPoint = new Point(row, col);
            setButtonEnabled(buttonTexts.SelectStart, !selected);
            setButtonEnabled(buttonTexts.SelectEnd, !selected);
            repaintOnePixel(clickedPoint, MazeConstants.TemporaryPoint);
        }
    }
}
```

Konstruktor inicjalizuje pola, ustawia tło panelu na białe, ustala rozmiar komórek labiryntu i dodaje **MouseListener**, aby obsłużyć kliknięcia myszką.

```

// Metoda ustawiajaca nowy punkt startowy
public void setNewStartPoint() { usage
    if (clickedPoint != null) {
        newStartPoint = clickedPoint;
        repaintOnePixel(parser.getStart(), MazeConstants.Path);
        parser.resetStart(newStartPoint);
        repaintOnePixel(parser.getStart(), MazeConstants.Start);
        setButtonEnabled(buttonTexts.SelectStart, isEnabled: false);
        setButtonEnabled(buttonTexts.SelectEnd, isEnabled: false);
    }
}

// Metoda ustawiajaca nowy punkt koncowy
public void setNewEndPoint() { usage
    if (clickedPoint != null) {
        newEndPoint = clickedPoint;
        repaintOnePixel(parser.getEnd(), MazeConstants.Path);
        parser.resetEnd(newEndPoint);
        repaintOnePixel(parser.getEnd(), MazeConstants.End);
        setButtonEnabled(buttonTexts.SelectEnd, isEnabled: false);
        setButtonEnabled(buttonTexts.SelectStart, isEnabled: false);
    }
}

```

Ustawia nowy punkt początkowy/końcowy na podstawie ostatnio klikniętego punktu. Aktualizuje wygląd labiryntu i wyłącza odpowiednie przyciski.

```

// Metoda do odświeżania jednego piksela
public void repaintOnePixel(Point p, char symbol) { usage
    int row = (int)p.getY();
    int col = (int)p.getX();

    var g = cachedImage.getGraphics();
    g.setColor(MazeConstants.getColor(symbol));
    g.fillRect(col * cellSize, (row * cellSize), cellSize, cellSize);
    g.setcolor(Color.BLACK);
    g.drawRect((col * cellSize), (row * cellSize), cellSize, cellSize);
    g.dispose();
    repaint();
}

// Metoda do ładowania labiryntu
public void loadMaze(MazeParser parser) { usage
    this.parser = parser;
    mazeService.createCachedImage();
    setPreferredSize(new Dimension(width: parser.getcols() * cellSize, height: parser.getRows() * cellSize));
    revalidate(); // Konieczne uaktualnienie rozmiaru
    repaint();
    setButtonEnabled(buttonTexts.FindPath, isEnabled: true);
}

```

repaintOnePixel(Point p, char symbol): Aktualizuje wygląd jednej komórki labiryntu na podstawie podanego symbolu. **loadMaze(MazeParser parser):** Wczytuje labirynt do panelu, tworzy jego buforowany obraz, aktualizuje rozmiar panelu i ustępuje przycisk do znajdowania ścieżki jako aktywny.

```

// Metoda do rozwiązywania labiryntu
public void solveMaze() { usage
    if (parser != null) {
        solveService.getSolvePoints(parser); // Zapisanie ścieżki rozwiązania
        ankietyk = mazeService.get_solvePoints(parser);
        for (Point solvePoint : solveSteps) {
            if ((solvePoint.equals(parser.getStart()) || solvePoint.equals(parser.getEnd())))
                repaintOnePixel(solvePoint, MazeConstants.Solution);
        }
        setButtonEnabled(buttonTexts.Export, isEnabled: true);
        setButtonEnabled(buttonTexts.FindPath, isEnabled: false);
        blokada = true;
    }
}

// Metoda do tworzenia buforowanego obrazu labiryntu
private BufferedImage createCachedImage() { usage
    BufferedImage image = new BufferedImage(parser.getcols() * cellSize, parser.getRows() * cellSize,
        BufferedImage.TYPE_32BPP);
    Graphics2D g = image.createGraphics();
    if (parser != null) {
        int rows = parser.getRows();
        int cols = parser.getcols();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                g.setColor(MazeConstants.getColor(parser.getMaze()[i][j]));
                g.fillRect(j * cellSize, (i * cellSize), cellSize, cellSize);
                g.setcolor(Color.BLACK);
                g.drawRect(j * cellSize, (i * cellSize), cellSize, cellSize);
            }
        }
        g.dispose();
    }
    return image;
}

```

solveMaze(): Rozwiązuje labirynt za pomocą serwisu **mazeService** i wyświetla ścieżkę rozwiązania. Po rozwiązaniu blokuje możliwość dalszych kliknięć i aktywuje przycisk eksportu.

`createCachedImage()`: Tworzy buforowany obraz labiryntu.

```
// Metoda do włączania i wyłączania przycisków
private void setButtonEnabled(String text, boolean isEnabled) { 9 usages
    for (int i = 0; i < buttons.size(); i++) {
        if (buttons.get(i).getText().equals(text)) {
            buttons.get(i).setEnabled(isEnabled);
        }
    }
}

// Metoda do rysowania komponentu
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    if (cachedImage != null) {
        g.drawImage(cachedImage, -offsetX, -offsetY, observer);
    }
}

// Metoda do przewijania widocznego obszaru
@Override
public void scrollRectToVisible(Rectangle aRect) {
    super.scrollRectToVisible(aRect);
    offsetX = aRect.x;
    offsetY = aRect.y;
    repaint();
}
```

`setButtonEnabled(String text, boolean isEnabled)`: Ustawia stan aktywności przycisków interfejsu na podstawie ich tekstu.

`paintComponent(Graphics g)`: Przesłonięta metoda do rysowania komponentu. Rysuje buforowany obraz labiryntu.

`scrollRectToVisible(Rectangle aRect)`: Aktualizuje przesunięcia i odświeża widok podczas przewijania.

3.4 Moduł przycisków interfejsu (gui.buttons)

Definiują przyciski o specyficznych funkcjach, takich jak ustawianie punktów startowych i końcowych, ładowanie i zapisywanie labiryntu.

3.4.1 ButtonTexts

```
15 usages
public class ButtonTexts {
    4 usages
    public static final String SelectStart = "Select start";
    4 usages
    public static final String SelectEnd = "Select end";
    3 usages
    public static final String FindPath = "Find path";
    1 usage
    public static final String LoadT = "Load";
    2 usages
    public static final String Export = "Save";
}
```

Zawiera stałe tekstowe używane jako etykiety dla przycisków w aplikacji.

3.4.2 LoadButton

```
public class Loadbutton extends JButton implements ActionListener {  
    private MazeFrame mazeFrame; // Reference do g³osnika labiryntu z oknem  
  
    // konstruktor klasy Loadbutton, przyjmuj±c jako argument  
    public Loadbutton(MazeFrame mazeFrame) { this.mazeFrame = mazeFrame; }  
  
    // metoda wywo³ana po klikni±ciu przycisku  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        JFileChooser fileChooser = new JFileChooser("C:/"); // Ustalenie sk³adki pliku  
        int returnvalue = fileChooser.showOpenDialog(null); // Ustalenie sposobu otwarcia i zamkniecia okna wyboru  
        if (returnvalue == JFileChooser.APPROVE_OPTION) {  
            File selectedFile = fileChooser.getSelectedFile(); // Wybrany ekran do odczytu  
            try {  
                MazeParser parser = new MazeParser(selectedFile); // Ustalenie obiektu MazeParser na podstawie wybranego pliku  
                MazeParser selectedFile; // Ustalenie obiektu MazeParser  
            } catch (Exception ex) {  
                ex.printStackTrace();  
                // Wyk³adanie komunikatu o b³±dzie w programie i zamkniecie okna  
                JOptionPane.showMessageDialog(null, "Error reading maze file!");  
            }  
        }  
    }  
}
```

Implementuje interfejs **ActionListener** i jest u¿ywana do obs³ugi przycisku ładowania pliku tekstowego z labiryntem.

3.4.3 SaveButton

```
public class Savebutton implements ActionListener {  
    private MazeFrame mazeFrame; // Reference do g³osnika labiryntu z oknem  
    private IMazeService mazeService; // Reference do serwisu  
  
    // konstruktor Savebutton, stworzy przyj±ty obiekt klasy MazeFrame.  
    public Savebutton(MazeFrame mazeFrame, IMazeService service) {  
        this.mazeFrame = mazeFrame;  
        mazeService = service;  
    }  
  
    // Metoda wywo³ana po klikni±ciu przycisku  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        FileNameExtensionFilter txtFilter = new FileNameExtensionFilter("Txt file (.txt)", ".txt");  
        FileNameExtensionFilter pngFilter = new FileNameExtensionFilter("Photo (.png)", ".png");  
        FileNameExtensionFilter binFilter = new FileNameExtensionFilter("Binary (.bin)", ".bin");  
  
        JFileChooser fileChooser = new JFileChooser();  
        fileChooser.setDialogTitle("Save Maze");  
        fileChooser.addChoosableFileFilter(txtFilter);  
        fileChooser.addChoosableFileFilter(pngFilter);  
        fileChooser.addChoosableFileFilter(binFilter);  
        fileChooser.setAcceptAllFileFilterUsed(false);  
        mazeService.saveMaze(fileChooser.getSelectedFile(), mazeFrame.getSaveSteps(), mazeFrame.getCachedImage());  
    }  
}
```

Implementuje interfejs **ActionListener** i jest u¿ywana do obs³ugi przycisku zapisu labiryntu w ró¿nych formatach.

3.4.4 SetEndButton

```
public class SetEndbutton extends JButton {  
    private MazeFrame frame; // Reference do g³osnika labiryntu z oknem  
  
    // konstruktor klasy SetEndbutton, przyjmuj±c MazeFrame i tekst przechowywany tam w argumenty  
    public SetEndbutton(MazeFrame frame, String text) {  
        super(text); // ustalenie tekstu przycisku  
        this.frame = frame; // Przechowanie referencji do MazeFrame  
    }  
  
    // Nadpisana metoda fireActionPerformed, ktora jest wywo³ana podczas klikni±cia przycisku  
    @Override  
    protected void fireActionPerformed(ActionEvent event) {  
        // Wyk³adanie nietypowej metody ustalania nowego punktu koncowego w MazeFrame  
        frame.setEndPoint();  
  
        // Wyk³adanie metody nadredefiniowanej aby zapewni³ standardowe zachowanie, takie jak powiadomianie o aktualizacji  
        super.fireActionPerformed(event);  
    }  
}
```

Rozszerza **JButton** i jest u¿ywana do ustawiania punktu końcowego w labiryncie.

3.4.5 SetStartBtn

```
public class SetStartBtn extends JButton {  
    private MazeFrame frame; // referencja do głównego ramki labiryntu z użyciem  
    // konstruktor klasy SetStartBtn, przyjmujący MazeFrame i tekst przycisku jako argumenty  
    public SetStartBtn(MazeFrame frame, String text) {  
        super(text); // ustawienia tekstu przycisku  
        this.frame = frame; // przechowanie referencji do MazeFrame  
    }  
  
    // Nadpisana metoda fireActionPerformed, która jest wywoływana podczas kliknięcia przycisku  
    @Override // użycie  
    protected void actionPerformed(ActionEvent event) {  
        // Wywołanie metod startSelectedPoint() metody ustanawiającej nowego punktu startowego w MazeFrame  
        frame.setStartPoint();  
  
        // Wyświetl informację o dodaniu, aby zapewnić standardowe zachowanie, takie jak ponadudnianie nastuchionego  
        super.fireActionPerformed(event);  
    }  
}
```

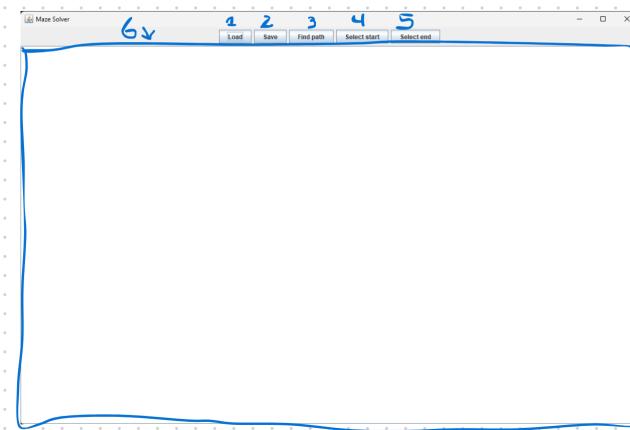
Rozszerza JButton i jest używana do ustawiania punktu startowego w labiryncie.

4 Opis interfejsu

Graficzny interfejs użytkownika jest wykonany w Javie z wykorzystaniem biblioteki Swing. Interfejs graficzny składa się z:

- paska menu z opcją do: wczytania labiryntu z pliku tekstowego i binarnego i zapisu labiryntu wraz z ew. rozwiązaniem,
- komponentu prezentującego wczytany labirynt,
- dodatkowego panelu narzędziowego, z przyciskami do: wyszukiwania najkrótszej ścieżki w labiryncie, zaznaczania nowych punktów startowych/końcowych przez wybranie ich myszką.

5 Wygląd interfejsu

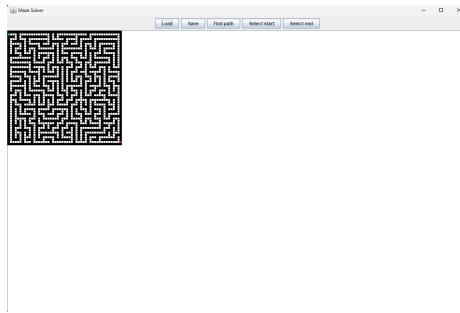


6 Opis elementów interfejsu

Interfejs składa się z sześciu komponentów.

6.1 Przycisk Load

Przy naciśnięciu otwiera się dialogowe okno w którym użytkownik wybiera plik do wczytania. Dostępne rozszerzenia do wyboru : .txt, .bin. Po wybraniu pliku program wyświetla labirynt w komponencie 6. Przycisk



6.2 Save

Przycisk daje możliwość zapisania labiryntu w postaci binarnej, tekstowej oraz png. Jest dostępny dopiero po znalezieniu najkrótszej ścieżki, czyli po naciśnięciu przyciska 3. Po naciśnięciu przyciska wyskakuje okno w którym użytkownik wpisuje nazwę pliku razem z rozszerzeniem.

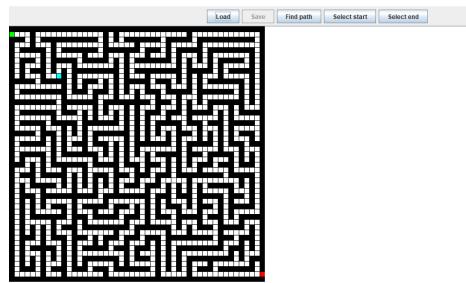
6.3 Find Path

Po naciśnięciu rysuje rozwiązańe labiryntu w komponencie 6. Szlag jest zaznaczony niebieskim kolorem.

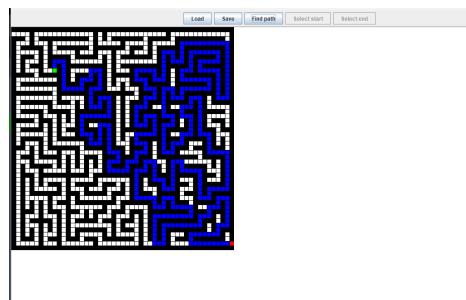


6.4 Select start

Przy naciskaniu na canvas kwadracik zmienia swój kolor na niebieski, co oznacza że jest zaznaczony i użytkownik może wybrać go jako start, korzystając z przycisku Select Start. Po naciśnięciu, wybrany kwadrat znowu zmienia swój kolor, tym razem na zielony, co oznacza że jest początkiem labiryntu. Także po naciśnięciu przycisk jest zablokowany i jest znowu dostępny po wybraniu nowego punktu.



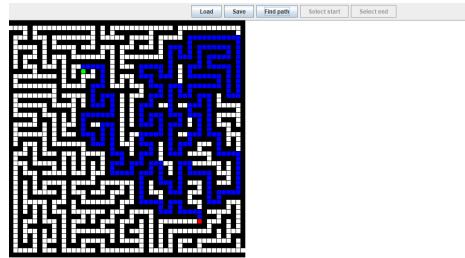
Wybieranie punktu



Znalezienie drogi z nowym startem

6.5 Select End

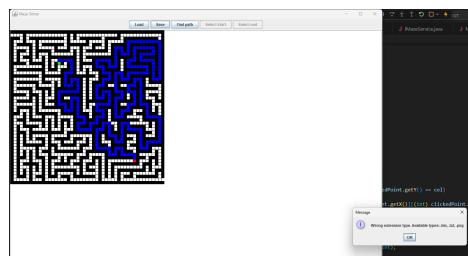
Działa analogicznie jak Select Start. Przycisk



Przykład działania

7 Komunikaty błędów

Wszystkie komunikaty są wyświetlane przy pomocy dialogowych okien , w których jest opisany problem.



Przykład komunikatu błędu

8 Diagram klas

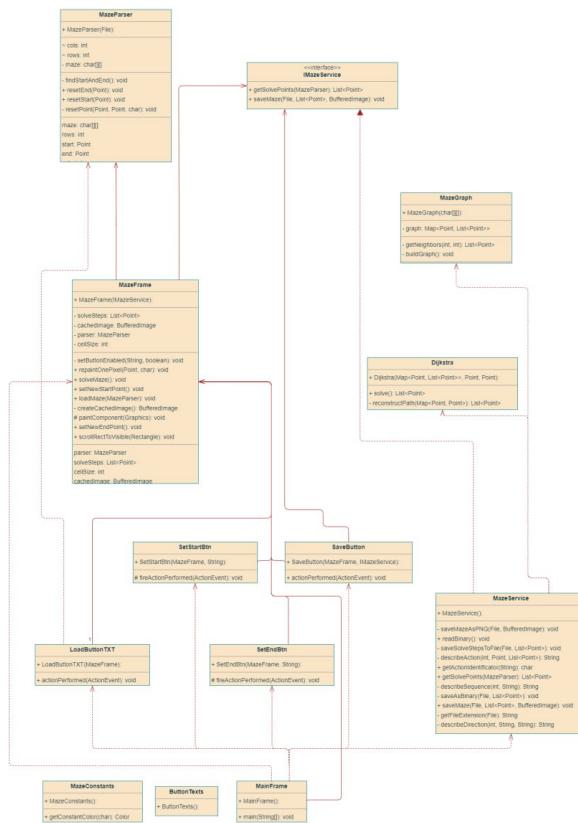


Diagram klas jest stworzony przy pomocy narzędzi pomocniczych