Project report

Pacman

Name : Trần Đức Hải Triều

ID: ITITWE 20027

Content

Chapter 1: Introduce

1.1 Overview

The Pacman Game Project is a project which aims to develop an interactive java application meant to relive the arcade classic game in a more beautiful and simplified format. It serves as a learning project as it shows the syntheses of basic programming skills, concepts of object oriented programming as well as the construction of a graphical user interface (GUI).

The game has a very simple interface that allows a player to roam about a maze collecting points and trying to avoid ghosts. The player can select from three levels of difficulty – Easy, Normal and Hard where the game's challenge increases at every selection in terms of speed of movement of the ghosts. Pacman's game centrals including the collision detection, ghost logic and Pacman's moves are coded Java-based Swing library as well as using game timers.

This project implements modular programming with project structure that consists primarily of two blocks: Main.java, which is responsible for the graphical interface of the game and its start, and Model.java, where the mathematics behind the game and its graphical display are contained.

1.2 Objectives

The Pac-Man Game Project has a primary goal of providing practical skills of software development with Java. This project is meant to achieve the following objectives: Improving Programming Skills, Comprehending the aspects of Game Development, Developing Problem Solving and Design Strategies, Creating an Entertaining Application. Upon achieving the set objectives, the project seeks to build the technical skills of the developer illustrating the application of programming language concepts in practice.

Chapter 2 software implementation

2.1 Development Environment:

The tools and technologies used in the development of the Pacman Game Project are as follows:

1.Programming Language:

+Java: A major programming language that was used for implementing the game logic, user interface and event handling.

2.Integrated Development Environment (IDE):

+IntelliJ IDEA/Eclipse: Intended to write, debug and handle the code in a better manner.

3.Libraries and Frameworks:

+Swing: This framework was required for constructing the graphical user interface (GUI) such as windows, buttons and panels.

+AWT: This was required for graphics rendering, events handling and layouts management.

+Timer: This was needed in the game for animations and updates at regular intervals.

4.System Requirements:

+Operating system: Will work with Windows, Mac OS, Linux.

+Java Development Kit (JDK): Required version 8 or higher for compiling as well as for running the program.

This platform came very useful in the development and debugging of the Pacman game while maintaining the functionality as well as the scope of development.

2.2 Core Features:

The Pac-Man Game Project includes the following core features that ensure an engaging and interactive gameplay experience:

1. Main Menu

- A user-friendly interface that provides:
    - **Start Game**: This option is for the player to start the game.
    - **Difficulty Selection**: choose one of the three difficulty levels, and it can determine the ghost movement speed.
    - **Exit Game**: Closes the application safely.

2. Game Mechanics

- **Pac-Man Movement**:
    - Controlled using keyboard arrow keys for smooth navigation through the maze.
    - Restricted movement to prevent passing through walls.
- **Ghost AI**:
    - Ghosts move autonomously within the maze.
    - Their speed and behavior vary based on the selected difficulty level.
- **Collision Detection and lives system**:
    - Each collision with a ghost reduces Pac-Man's lives. Players begin with three lives, and the game ends when all lives are lost

3. Scoring System

- Points are earned by collecting food pellets scattered across the maze. Every pellet Pac-Man consumes not only increases the score but also clears the

path, pushing players closer to victory. This feature encourages players to strategize their movements for maximum points while avoiding danger. Collected items disappear, updating the game map dynamically.

4. Graphics and Animations

- **Maze Design**:
    - The maze is visually represented using a 2D grid structure with distinct colors for walls, food, and empty spaces.
- **Character Rendering**:
    - Pac-Man and ghosts are rendered with distinct visual designs.
    - Real-time animations enhance the game experience.

5. Difficulty Levels

- **Easy**: Ghosts move slowly, providing a more relaxed experience.
- **Normal**: Ghosts move at moderate speed, offering a balanced challenge.
- **Hard**: Ghosts move quickly, creating a more intense and competitive environment.

6. Dynamic Gameplay

- Continuous game updates using timers for smooth animations and interactions.
- Allows players to strategize their moves based on ghost positions and the maze layout.

2.3 Modular Structure:

1. Main Module (Main.java)

- **Purpose**: Provides as the entry point for the application.
- **Responsibilities**:
    - Creates and manages the main menu interface using Java Swing components.
    - Handles user interactions for starting the game, selecting difficulty levels, and exiting the application.
    - Transitions from the main menu to the game screen upon user input.
- **Key Components**:
    - **Main Menu**: Includes buttons for "Start Game," "Exit," and a difficulty selection dialog.
    - **Event Handling**:
      Listens for user input using action listeners and acts accordingly.

2. Game Logic Module (Model.java)

- **Purpose**: The core game logics and rendering are implemented here.
- **Responsibilities**:
    - Keeps track of the game state, such as the movement of Pac-Man, the behavior of ghosts, and collision detection.
    - Updates the game map dynamically based on the interaction of Pac-Man with food items and ghosts.
    - Implements the scoring and life systems.
- **Key Components**:
    - **Map Initialization**: A 2D array represents the maze, including walls, food items, and empty spaces.
    - **Character Behavior**: Defines Pac-Man's movement based on keyboard inputs and ghost behavior using AI-like patterns.

- **Collision Detection**: Ensures proper interaction between Pac-Man, ghosts, and the maze elements.
- **Game Timer**: Regularly updates the game state to create smooth animations and dynamic interactions.

3. Supporting Features

- **Difficulty Adjustment**:
  - Ghost speed dynamically adjusts based on the selected difficulty level.
  - Ensures scalability for players with varying skill levels.
- **Event Handling**:
  - Key listeners manage Pac-Man's movement.
  - Timers handle real-time updates for smooth animations and ghost movement.

4. Integration

- **Communication Between Modules**:
  - The Main.java module starts the game by instantiating the Model class according to user input.
  - The Model.java module manages the game logics while reporting the game status, such as game over, to the main module for proper actions.

3. Design and Implementation

3.1 User Interface Design:
The user interface (UI) of the Pac-Man Game Project is designed to be simple, intuitive, and visually appealing. It ensures a seamless interaction between the player and the game while maintaining the nostalgic essence of the classic Pac-Man.

1. Main Menu

The main menu is the starting point of the game, offering easy navigation and clear options:

- **Components**:
    - **Title Label**: A prominently displayed "Pac-Man" title with a bold font to capture the player's attention.
    - **Buttons**:
        - **Start Game**: Launches the game with the selected difficulty.
        - **Exit**: Closes the application.
    - **Difficulty Selection Dialog**: Opens a pop-up allowing the player to choose between Easy, Normal, and Hard modes.
- **Layout**:
    - A centered vertical layout for intuitive navigation.
    - Styled with spacing and alignment to ensure a visually balanced design.

2. Game Screen

The game screen is where the core gameplay takes place, offering an immersive experience:

- **Components**:
    - **Maze**:
        - Designed as a 2D grid with distinct visual elements for walls, food items, and empty spaces.

        - Walls are represented as blue blocks, food items as yellow dots, and pathways are left blank.
    - **Characters**:
        - **Pac-Man**: Rendered as an orange circle for simplicity and visibility.
        - **Ghosts**: Displayed as red circles with smooth movement animations.
    - **Status Bar**:
        - Displays the player's remaining lives in a corner for easy visibility.
        - Updates dynamically as the game progresses.
- **Layout**:
    - The maze occupies the central portion of the screen, ensuring focus on gameplay.
    - The status bar is unobtrusive but accessible, providing essential information without cluttering the interface.

3. Interaction Design

The UI is highly interactive, responding smoothly to user inputs:

- **Keyboard Controls**:
    - Arrow keys are used to navigate Pac-Man through the maze.
- **Difficulty Dialog**:
    - Pop-up dialog for difficulty selection ensures clarity and prevents accidental selections.
- **Game Over Notification**:
    - A dialog box appears when the game ends, displaying a "Game Over" message and allowing the player to exit or restart.

4. Visual and Aesthetic Elements

- **Color Scheme**:
    - Bright, contrasting colors (e.g., blue walls, yellow food, red ghosts, orange Pac-Man) enhance visibility and gameplay experience.
- **Fonts**:
    - Clean, bold fonts for labels and buttons ensure readability.
- **Animation**:
    - Smooth animations for character movement and interactions add to the dynamic feel of the game.

5. Accessibility

The UI is designed to be accessible and straightforward for players of all ages:

- Clear labeling of buttons and dialogs.
- Simplified navigation to minimize confusion.
- Consistent visual elements for intuitive gameplay.

3.2 Game Logic Design

```
public class Main {
    public static void main(String[] args) {
        JFrame frame = new JFrame( title: "Pac Man");
        frame.setVisible(true);
        frame.setSize( width: 400,  height: 500);
        frame.setLocationRelativeTo(null);
        frame.setResizable(false);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel panel = new JPanel();
        panel.setLayout(new GridBagLayout());
        frame.add(panel);

        JPanel innerPanel = new JPanel();
        innerPanel.setLayout(new BoxLayout(innerPanel, BoxLayout.Y_AXIS));
        panel.add(innerPanel);

        JLabel titleLabel = new JLabel( text: "Pacman");
        titleLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
        titleLabel.setFont(new Font( name: "Arial", Font.BOLD,  size: 80));
        innerPanel.add(titleLabel);

        JButton startButton = new JButton( text: "Start Game");
        startButton.setAlignmentX(Component.CENTER_ALIGNMENT);
        startButton.addActionListener(e -> showDifficultyDialog(frame));
        innerPanel.add(startButton);

        JButton exitButton = new JButton( text: "Exit");
        exitButton.setAlignmentX(Component.CENTER_ALIGNMENT);
        exitButton.addActionListener(e -> System.exit( status: 0));
        innerPanel.add(exitButton);
    }
```

The application consists of a JFrame titled "Pac Man" with a fixed size of 400x500 pixels. Inside the JFrame, there is a vertically aligned panel containing a title label ("Pacman") and two buttons: "Start Game" and "Exit."

The "Start Game" button is designed to trigger a method (showDifficultyDialog) for selecting a game difficulty, while the "Exit" button closes the application. The layout ensures all components are centered and arranged vertically for a clean, user-friendly design.

```
private static void showDifficultyDialog(JFrame parentFrame) {  1 usage
    JDialog dialog = new JDialog(parentFrame,  title: "Select Difficulty",  modal: true);
    dialog.setSize( width: 300,  height: 200);
    dialog.setLocationRelativeTo(parentFrame);

    JButton easyButton = new JButton( text: "Easy");
    easyButton.addActionListener(e -> startGame(parentFrame,  difficulty: 1));
    JButton normalButton = new JButton( text: "Normal");
    normalButton.addActionListener(e -> startGame(parentFrame,  difficulty: 2));
    JButton hardButton = new JButton( text: "Hard");
    hardButton.addActionListener(e -> startGame(parentFrame,  difficulty: 3));

    JPanel buttonPanel = new JPanel();
    buttonPanel.setLayout(new GridLayout( rows: 3,  cols: 1));
    buttonPanel.add(easyButton);
    buttonPanel.add(normalButton);
    buttonPanel.add(hardButton);

    dialog.add(buttonPanel);
    dialog.setVisible(true);
}
```

The showDifficultyDialog method creates a modal dialog to allow users to select a game difficulty level. The dialog contains three buttons: "Easy," "Normal," and "Hard," each triggering the startGame method with a respective difficulty value (1, 2, or 3). The buttons are arranged vertically within a panel using a GridLayout, and the dialog is displayed centered relative to the parent frame.

```
@     private static void startGame(JFrame parentFrame, int difficulty) {  3 usages
          parentFrame.dispose();
          JFrame gameFrame = new JFrame( title: "PacMan - Playing");
          Model game = new Model(difficulty);
          gameFrame.add(game);
          gameFrame.setSize( width: 400,  height: 500);
          gameFrame.setLocationRelativeTo(null);
          gameFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          gameFrame.setVisible(true);
      }
  }
```

The startGame method initializes and displays a new game window. It closes the parent frame and creates a new JFrame titled "PacMan - Playing." A Model object (likely representing the game logic) is added to the frame with the selected difficulty. The game window is set to a fixed size of 400x500 pixels, centered on the screen, and configured to close the application on exit.

2. Logic:

```
class Model extends JPanel {  2 usages

    private int[][] map;  13 usages
    private int tileSize = 20;  18 usages
    private int pacmanX, pacmanY; // Pac-Man's position  8 usages
    private List<Ghost> ghosts; // List of ghosts  6 usages
    private int ghostSpeed = 1; // Speed of ghost movement  3 usages
    private int lives = 3; // Number of Pac-Man lives  5 usages
    private Timer timer; // Timer for game updates  2 usages

    private Image pacmanImage; // Pac-Man image  no usages
    private Image ghostImage;  // Ghost image  no usages

    public Model(int difficulty) {  1 usage

        // Adjust ghost speed based on difficulty
        ghostSpeed = difficulty;


        // Initialize the game map
        initializeMap();

        // Place Pac-Man at the starting position
        pacmanX = 1;
        pacmanY = 1;

        // Initialize ghosts
        initializeGhosts();

        // Add a key listener for movement
        setFocusable(true);
```

- **Attributes**: The class contains variables for the game map, Pac-Man's position, ghost properties (speed and list), lives, a timer for updates, and images for Pac-Man and ghosts.
- **Constructor**:
    - Initializes the game with the given difficulty, setting the ghost speed accordingly.
    - Calls methods to initialize the map (initializeMap) and ghosts (initializeGhosts).
    - Sets Pac-Man's starting position at (1, 1).
    - Enables the panel to listen for keyboard input with setFocusable(true).

```
    // Add a key listener for movement
    setFocusable(true);
    addKeyListener((KeyAdapter) keyPressed(e) -> {
            movePacman(e.getKeyCode());
            repaint();
    });

    // Start the game timer
    timer = new Timer( delay: 200, e -> {
        moveGhosts();
        checkCollisions();
        repaint();
    });
    timer.start();

}
```

Keyboard Input for Pac-Man:

- A KeyListener is added to detect keyboard events.
- When a key is pressed, movePacman(e.getKeyCode()) is called to move Pac-Man based on the key pressed.
- repaint() is called to refresh the game display after movement.

Game Timer:

- A Timer is initialized with a delay of 200 milliseconds, executing periodic game updates.
- Within each timer tick:
    - moveGhosts() moves the ghosts.
    - checkCollisions() checks for collisions (likely between Pac-Man and ghosts or map elements).
    - repaint() refreshes the game display.

Starting the Timer:

- timer.start() starts the periodic updates.

```
private void initializeMap() {  1 usage
    // 0 = empty space, 1 = wall, 2 = food
    map = new int[][] {
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
            {1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1},
            {1, 2, 1, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 2, 1, 1, 1, 1, 2, 1},
            {1, 2, 1, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 1, 2, 1},
            {1, 2, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 2, 1, 1, 2, 1},
            {1, 2, 2, 2, 2, 2, 2, 1, 2, 1, 2, 1, 2, 2, 2, 2, 2, 2, 2, 1},
            {1, 1, 1, 1, 1, 1, 2, 1, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1},
            {1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 1, 2, 2, 2, 2, 2, 2, 2, 1},
            {1, 2, 1, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 2, 2, 2, 1, 2, 1},
            {1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 2, 1},
            {1, 2, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 1, 2, 1},
            {1, 2, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1},
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
    };
}
```

The initializeMap method defines the layout of the game map using a 2D array. Each integer in the array represents a different type of cell in the game:

- 1: Wall that blocks movement.
- 2: Food that Pac-Man can eat and move.

The map array is structured as a grid where each row represents a horizontal slice of the game map. This layout creates a maze-like structure for the game, with walls (1) forming boundaries and paths, and food (2) scattered throughout the playable areas.

```java
private void initializeGhosts() {  1 usage
    ghosts = new ArrayList<>();
    ghosts.add(new Ghost( x: 9,  y: 6)); // Add a ghost at position (9, 6)
    ghosts.add(new Ghost( x: 10,  y: 10)); // Add another ghost
}

private void movePacman(int keyCode) {  1 usage
    int newX = pacmanX;
    int newY = pacmanY;

    switch (keyCode) {
        case KeyEvent.VK_UP:
            newY--;
            break;
        case KeyEvent.VK_DOWN:
            newY++;
            break;
        case KeyEvent.VK_LEFT:
            newX--;
            break;
        case KeyEvent.VK_RIGHT:
            newX++;
            break;
    }
```

This code defines two key functionalities for the game:

1. **initializeGhosts Method**:
    - This method initializes the list of ghosts in the game.
    - Two ghost objects are added to the ghosts list at specific positions (x=9, y=6 and x=10, y=10).
    - The ghosts' positions are predefined and likely determine their starting points on the map.
2. **movePacman Method**:
    - This method handles Pac-Man's movement based on keyboard input.
    - The current position of Pac-Man (pacmanX and pacmanY) is used to calculate a new position (newX and newY).

- A switch statement checks the key pressed:
  - KeyEvent.VK_UP: Moves Pac-Man up by decrementing newY.
  - KeyEvent.VK_DOWN: Moves Pac-Man down by incrementing newY.
  - KeyEvent.VK_LEFT: Moves Pac-Man left by decrementing newX.
  - KeyEvent.VK_RIGHT: Moves Pac-Man right by incrementing newX.
- This logic determines Pac-Man's intended new position.

```
    // Check boundaries and collisions
    if (newX >= 0 && newX < map[0].length && newY >= 0 && newY < map.length && map[newY][newX] != 1) {
        pacmanX = newX;
        pacmanY = newY;

        // Eat food
        if (map[pacmanY][pacmanX] == 2) {
            map[pacmanY][pacmanX] = 0;
        }
    }
}
```

This code handles Pac-Man's movement logic, including collision checks and food consumption:

1. **Collision and Boundary Check**:
   - The if condition ensures Pac-Man's new position (newX, newY) is:
     - Within the map's boundaries.
     - Not colliding with a wall (map[newY][newX] != 1).
   - If the conditions are met, Pac-Man's position is updated to the new coordinates (pacmanX = newX; pacmanY = newY).
2. **Food Consumption**:
   - After moving, it checks if Pac-Man lands on a cell containing food (map[pacmanY][pacmanX] == 2).
   - If true, the food is "eaten" by setting that map cell to 0.

```
private void moveGhosts() {  1 usage
    for (Ghost ghost : ghosts) {
        int newX = ghost.x + ghost.dx * ghostSpeed;
        int newY = ghost.y + ghost.dy * ghostSpeed;

        // Change direction if ghost hits a wall
        if (newX < 0 || newX >= map[0].length || newY < 0 || newY >= map.length || map[newY][newX] == 1)
            ghost.changeDirection();
        } else {
            ghost.x = newX;
            ghost.y = newY;
        }

        // Check collision with Pac-Man
        if (ghost.x == pacmanX && ghost.y == pacmanY) {
            lives--;
            if (lives == 0) {
                JOptionPane.showMessageDialog( parentComponent: this,  message: "Game Over!");
                System.exit( status: 0);
            }
        }
    }
}
```

Move Each Ghost:For each ghost in the ghosts list, its new position (newX, newY) is calculated using its current position, direction (dx, dy), and speed (ghostSpeed).

Wall Collision Check:If the new position is out of bounds or hits a wall (map[newY][newX] == 1), the ghost changes direction by calling ghost.changeDirection().Otherwise, the ghost's position is updated to the new coordinates (ghost.x = newX; ghost.y = newY).

Collision with Pac-Man:If a ghost's position matches Pac-Man's position (ghost.x == pacmanX && ghost.y == pacmanY):Pac-Man loses a life (lives--).If lives reach zero, a "Game Over" dialog is displayed, and the program exits.

```
private void checkCollisions() {  1 usage
    for (Ghost ghost : ghosts) {
        if (ghost.x == pacmanX && ghost.y == pacmanY) {
            lives--;
            if (lives == 0) {
                JOptionPane.showMessageDialog( parentComponent: this,  message: "Game Over!");
                System.exit( status: 0);
            }
        }
    }
}
```

Loop Through Ghosts: It iterates through the ghosts list.

Collision Detection:For each ghost, it checks if the ghost's position matches Pac-Man's position (ghost.x == pacmanX && ghost.y == pacmanY).

Life Reduction:If a collision is detected, Pac-Man loses a life (lives--).

Game Over:If lives reaches zero, a "Game Over" dialog is displayed using JOptionPane.showMessageDialog, and the application exits with System.exit(0).

```java
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);

    for (int y = 0; y < map.length; y++) {
        for (int x = 0; x < map[0].length; x++) {
            if (map[y][x] == 1) {
                g.setColor(Color.BLUE);
                g.fillRect( x: x * tileSize,  y: y * tileSize, tileSize, tileSize);
            } else if (map[y][x] == 2) {
                g.setColor(Color.YELLOW);
                g.fillOval( x: x * tileSize + tileSize / 4,  y: y * tileSize + tileSize / 4,  width: tileSiz
            }
        }
    }

    // Draw Pac-Man
    g.setColor(Color.ORANGE);
    g.fillOval( x: pacmanX * tileSize,  y: pacmanY * tileSize, tileSize, tileSize);

    // Draw Ghosts
    g.setColor(Color.RED);
    for (Ghost ghost : ghosts) {
        g.fillOval( x: ghost.x * tileSize,  y: ghost.y * tileSize, tileSize, tileSize);
    }


    // Draw Lives
    g.setColor(Color.GREEN);
    g.drawString( str: "Lives: " + lives,  x: 10,  y: 20);
}
```

Call Parent Method: super.paintComponent(g); ensures the panel is properly rendered before custom drawing.

Draw Map:

Loops through the map array to determine the type of tile:

1 (Wall): Drawn as a blue rectangle (g.fillRect).

2 (Food): Drawn as a yellow oval (g.fillOval).

Draw Pac-Man:Sets the color to orange and draws Pac-Man as a filled oval at his current position (pacmanX, pacmanY) scaled by tileSize.

Draw Ghosts:

Sets the color to red and loops through the ghosts list.

Each ghost is drawn as a red filled oval at its position (ghost.x, ghost.y) scaled by tileSize.

Draw Lives: Sets the color to green and displays the remaining lives as a string in the top-left corner.

```
    // Inner class for Ghost
    class Ghost {  6 usages
        int x, y;  6 usages
        int dx = 0, dy = 1; // Initial direction  4 usages

        public Ghost(int x, int y) {  2 usages
            this.x = x;
            this.y = y;
        }

        public void changeDirection() {  1 usage
            // Randomize direction
            dx = (int) (Math.random() * 3) - 1;
            dy = (int) (Math.random() * 3) - 1;

            // Ensure ghosts don't stop moving
            if (dx == 0 && dy == 0) {
                dx = 1;
            }
        }
    }
}
```

Key Components:

Attributes:

x, y: The ghost's current position on the map.

dx, dy: The ghost's movement direction (default initialized to move downward).

Constructor: Ghost(int x, int y): Initializes the ghost's starting position using the provided x and y coordinates.

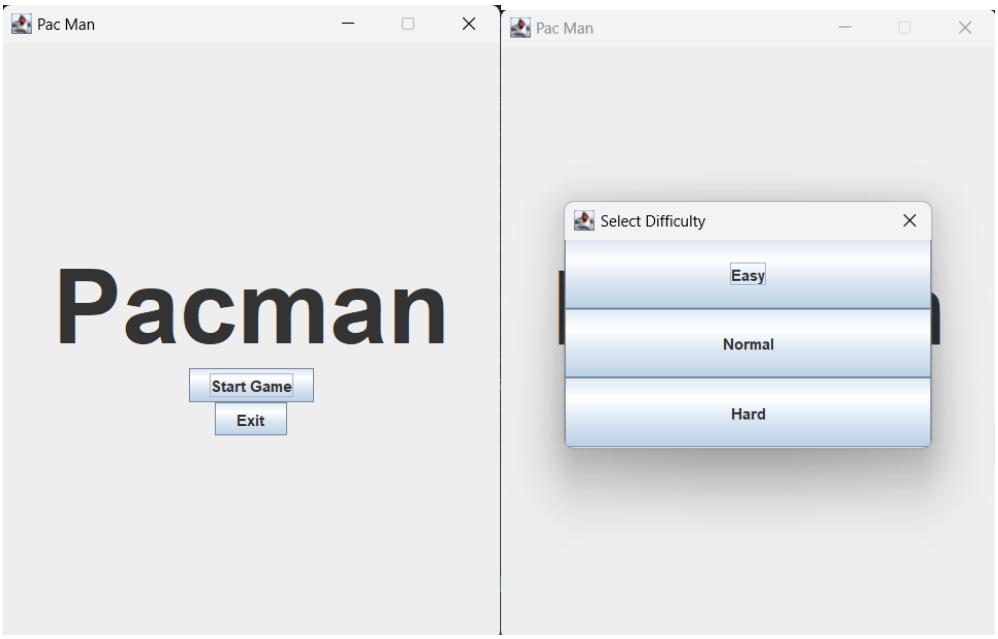
changeDirection Method:

Randomizes the ghost's movement direction: dx and dy are assigned random values between -1, 0, and 1 using Math.random().

Ensures the ghost continues moving by checking if both dx and dy are 0 (no movement). If so, dx is set to 1.

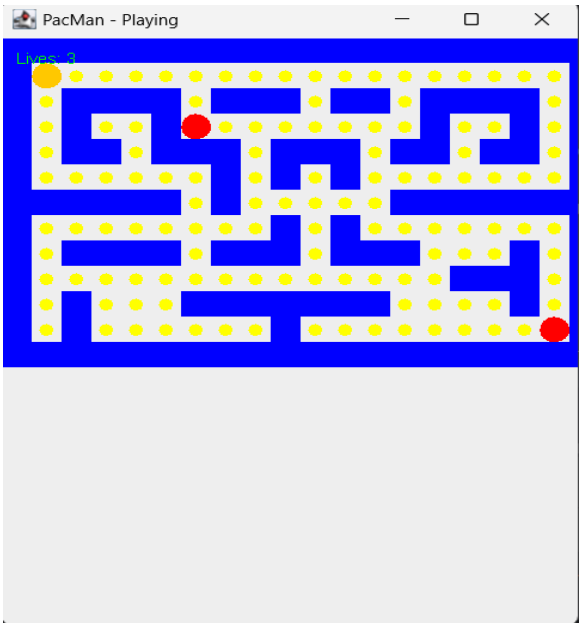
4. Application Gameplay:

4.1 main menu:



To play the Pac-Man game, start by clicking "Start Game" in the main menu and selecting your desired difficulty (Easy, Normal, or Hard)

4.2 In-Game Mechanics:



Objective:

- Collect all the yellow dots (food) on the board while avoiding the ghosts.
- You lose a life if a ghost touches Pac-Man. The game ends when all lives are lost.

Controls:

- Use the **arrow keys** to move Pac-Man:
    - Up Arrow: Move up.
    - Down Arrow: Move down.

- Left Arrow: Move left.
- Right Arrow: Move right.

Gameplay:

- Pac-Man can move through the maze but cannot pass through walls (blue blocks).
- Ghosts move randomly, and their speed depends on the selected difficulty level.
- The current number of lives is displayed at the top left.

Winning Condition:

- Eat all the yellow dots on the board to win the game.

Losing Condition:

- If a ghost catches Pac-Man, you lose a life. The game ends when all lives are depleted.

5. conclusion

This Pac-Man project is a simplified recreation of the classic arcade game, showcasing essential Java concepts such as object-oriented programming, GUI design with Swing, and event-driven programming. The game features dynamic gameplay with movable Pac-Man and ghosts, collision detection, and adjustable difficulty levels.

Key highlights include:

- Smooth user controls and visual appeal with images and color-coded mazes.
- Interactive gameplay with logical food collection and ghost avoidance.

This project serves as a solid foundation for 2D game development, offering room for enhancements like power-ups, advanced ghost AI, and a scoring system. It's a great starting point for learning Java-based game programming.