HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

PROJECT REPORT
OBJECT-ORIENTED PROGRAMMING

TOPIC: CIRCUIT PUZZLE GAME

(Circuit Simulation Construction)

Group: CyberSec223
Team: 26
Members:
1. Vu Dinh Nguyen - 20235612
2. Chu Van An - 20235581
3. Le Duy Anh - 20225565

Hanoi, 01/2026

Contents

1 Introduction

1.1 Overview

The **Circuit Puzzle** project is an educational simulation application designed to model basic electrical circuits. Built using Java and the Swing library, the application provides a "sandbox" environment where users can interactively assemble electronic components—such as power sources, resistors, bulbs, and wires—to create functioning circuits.

1.2 Project Objectives

The core objective of the system is to calculate physical properties of the circuit in real-time. The application aims to:

- Visualize electrical connectivity using a graph-based approach.
- Calculate **Voltage** (V), **Resistance** (R), and **Current** (I) using Ohm's Law.
- Provide visual feedback, such as lighting up bulbs or changing wire colors, when the circuit is complete.

2 System Design

2.1 Object-Oriented Architecture

The system relies on a robust Object-Oriented structure where all elements inherit from an abstract base class, `Component`. This design enforces a standard interface for drawing and electrical logic, ensuring that the simulation engine can treat all parts uniformly (Polymorphism).

2.1.1 The Abstract Component Class

The `Component` class defines the essential properties like coordinates (x, y), dimensions, and the rotation state. It also defines the contract for the `draw()` method, which every specific component must implement.

```java
public abstract class Component {
    protected int x, y;
    protected int width = 80, height = 40;
    protected int rotation = 0;
    protected boolean isPowered = false;
    protected String name;

    public Component(int x, int y, String name) {
        this.x = x;
        this.y = y;
        this.name = name;
    }

    // Abstract method forcing subclasses to define their own rendering
    public abstract void draw(Graphics2D g2);

    public void rotate() {
        rotation = (rotation + 1) % 4;
        int temp = width;
        width = height;
        height = temp;
    }

    public Rectangle getBounds() {
        return new Rectangle(x, y, width, height);
    }
}
```

Listing 1: The Abstract Component Class (Component.java)

2.2 Component Specialization

Concrete classes extend this base class to provide specific behaviors:

- **Battery:** Overrides `getValue()` to provide voltage.
- **Resistor:** Overrides `getValue()` to provide resistance (100Ω).
- **Bulb:** Implements specialized drawing logic to glow when powered.

3 Implementation Details

3.1 Simulation Algorithm (Graph Traversal)

To determine if a complete circuit exists, the system treats components as nodes in a graph. A Depth-First Search (DFS) algorithm, implemented in the `findConnectedPath` method, recursively finds all components physically connected to the power source.

```java
private void findConnectedPath(Component current, Set<Component> visited, List<Component> list) {
    visited.add(current);
    list.add(current);

    // Iterate through all components on the board
    for (Component neighbor : components) {
        // Check for physical overlap using bounding boxes
        if (!visited.contains(neighbor) && current.isTouching(neighbor)) {
            findConnectedPath(neighbor, visited, list);
        }
    }
}
```

Listing 2: Recursive Pathfinding Algorithm (CircuitBoard.java)

This algorithm ensures that only components actually wired to the Battery are activated.

3.2 Graphics and Rendering

The application utilizes `Graphics2D` for high-quality rendering. Components use `AffineTransform` to handle rotation around their center point.

For example, the `Resistor` class draws a standard zigzag symbol and applies a "Neon" color effect:

```java
@Override
public void draw(Graphics2D g2) {
    AffineTransform old = g2.getTransform();
    // Rotate the canvas around the component's center
    g2.rotate(Math.toRadians(rotation * 90), x + width/2, y +
   height/2);

    // Swap dimensions if rotated 90 or 270 degrees
    int w = (rotation % 2 == 0) ? width : height;
    int h = (rotation % 2 == 0) ? height : width;

    // Set Neon Magenta color
    g2.setColor(new Color(255, 0, 255));
    g2.setStroke(new BasicStroke(3));

    // Draw the characteristic Zigzag pattern
    int[] xp = {x+15, x+20, x+30, x+40, x+50, x+60, x+w-15};
    int[] yp = {y+h/2, y+h/2-10, y+h/2+10, y+h/2-10, y+h/2+10, y+h
   /2-10, y+h/2};
    g2.drawPolyline(xp, yp, xp.length);

    g2.setTransform(old); // Restore original rotation
}
```

Listing 3: Drawing Logic with Rotation (Resistor.java)

4 Product Images

This section provides a visual overview of the **Circuit Puzzle** application, demonstrating the user interface design, component aesthetics, and the real-time simulation feedback system.

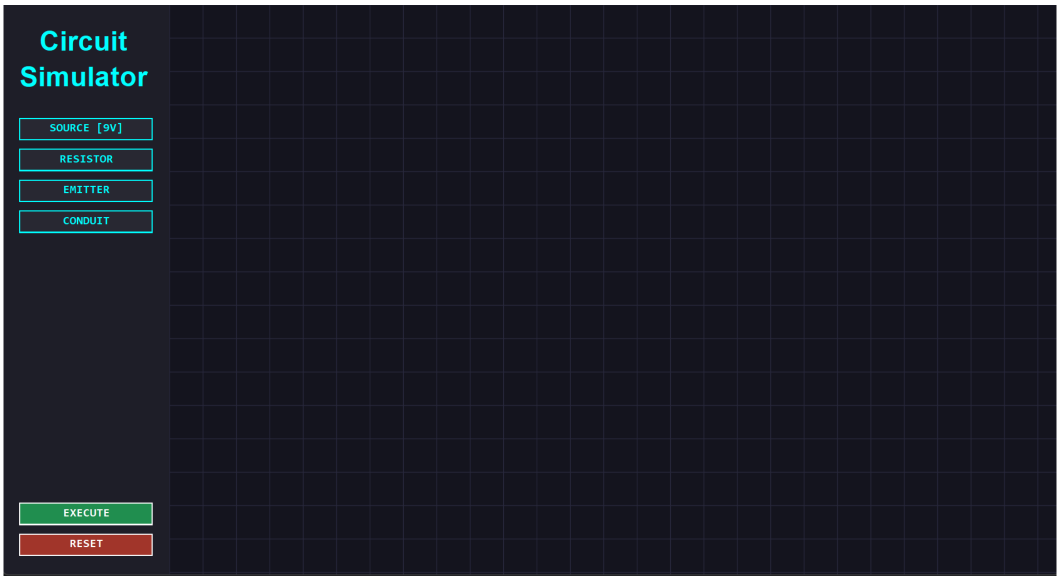


Figure 1: **Workspace Overview:** The application features a dark-themed, grid-based canvas designed for high contrast. The sidebar on the left provides quick access to the component library (Source, Resistor, Emitter, Conduit) and simulation controls.

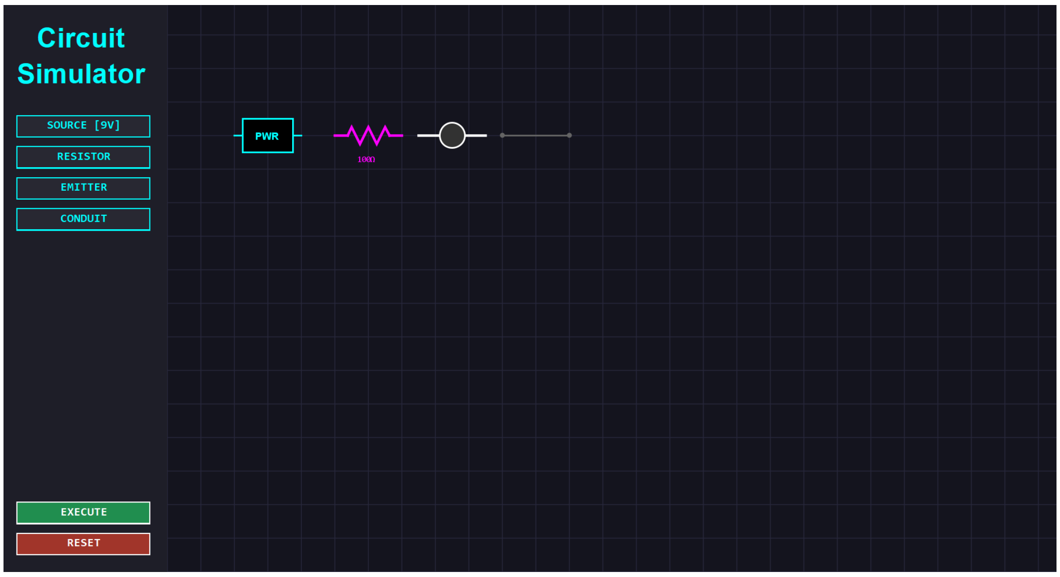
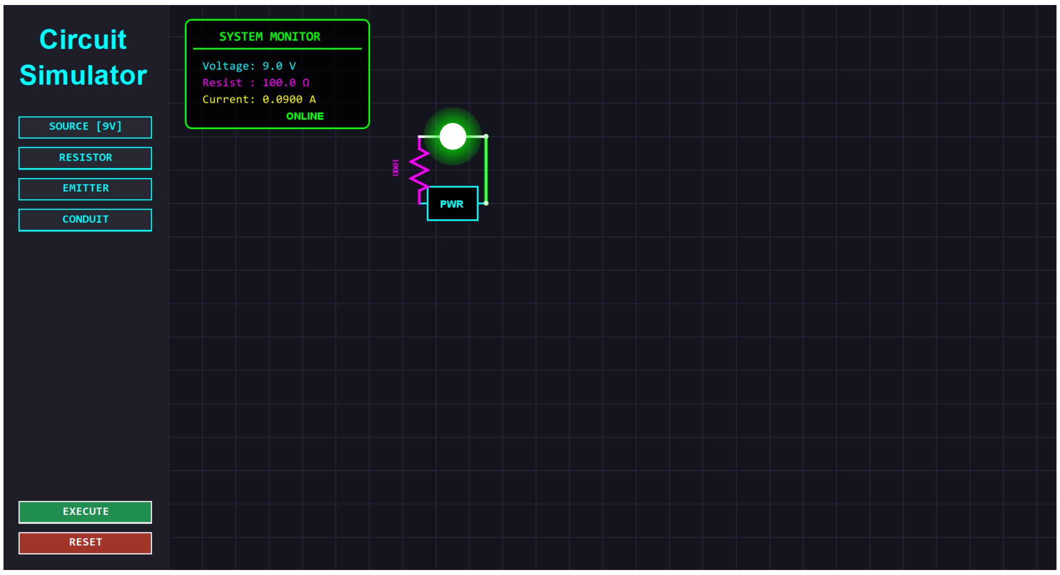


Figure 2: **Component Design:** Each electronic component is rendered with a distinct "Neon" style. The symbols (e.g., the zigzag for Resistors) follow standard electrical diagrams but are stylized to fit the "Cyberpunk" aesthetic of the project.



```
SYSTEM MONITOR
Voltage: 9.0 V
Resist : 100.0 Ω
Current: 0.0900 A
              ONLINE
```

Figure 3: **Live Simulation HUD:** When the simulation is executed, the **System Monitor** overlay appears, displaying calculated values for Voltage (V), Resistance (Ω), and Current (A). Visual cues, such as the glowing green bulb, indicate an active and valid circuit path.

5 Conclusion

5.1 Summary

The **Circuit Puzzle** project successfully demonstrates the application of Object-Oriented Programming principles in building a complex simulation tool. By leveraging inheritance, polymorphism, and encapsulation, the team created a scalable framework where new electrical components can be added with minimal changes to the core engine.

5.2 Future Enhancements

Future iterations of the software could include:

- **Parallel Circuits:** Solving Kirchhoff's laws for complex branching paths.
- **Save/Load System:** Serializing the component list to JSON or XML to save designs.
- **Advanced Components:** Adding Capacitors, Inductors, and Transistors.