# Circuit Puzzle Game (RC Timing & Drag-and-Drop)

Group 08

December 21, 2025

## Abstract

This report presents the design and implementation of a **Circuit Puzzle Game**, an educational JavaFX-based application that helps students understand basic electrical circuit concepts through interactive gameplay. Players construct electrical circuits by dragging components such as wires, resistors, and capacitors onto a board. The game evaluates the circuit behavior (e.g., total resistance, capacitance, and bulb lighting duration) and checks whether the player meets predefined objectives.

The project focuses on applying **Object-Oriented Programming (OOP)** principles including abstraction, inheritance, polymorphism, and modular design. UML use case and class diagrams are used to clearly describe system behavior and structure.

## I. Assignment of Members

### 1. Nguyen Hoang Anh

- **Student ID:** 202416773
- **Role:**
  - Requirement analysis
  - System design (Usecae diagram, Class Diagram)
  - Java implementation (UI, simulation)
  - Documentation and report writing

### 2. Pham Duy Nguyen

- **Student ID:** 202416730
- **Role:**
  - Requirement analysis
  - System design (Use Case Diagram, Class Diagram)
  - Java implementation (Game logic, UI, simulation)
  - Documentation and report writing

### 3. Do Thanh Trung

- **Student ID:** 202416833
- **Role:**
  - Requirement analysis
  - System design (Use Case Diagram)
  - Java implementation (Game logic, simulation)
  - Documentation and report writing

### 4. Le Viet Anh

- **Student ID:** 202416774
- **Role:**
  - Requirement analysis
  - System design (Use Case Diagram, Class Diagram)
  - Java implementation (Game component, simulation)
  - Documentation, report writing and powerpoint slide

**General:** All source code and ideas used in this project are either self-developed or adapted with understanding. No unauthorized copying was performed.

## II. Mini-Project Description

### 1. Project Requirements

The Circuit Puzzle Game must:

- Allow users to select a circuit level (Series or Parallel).
- Provide a toolbox containing electrical components.
- Enable drag-and-drop placement of components onto a grid board.
- Simulate circuit behavior based on placed components.
- Display circuit results ( Is the electrical panel working or not, and if it is, for how many seconds will the lights be on?)

### 2. Use Case Diagram and Explanation

#### 2.1. Actor

- **Player**: interacts with the system to solve circuit puzzles.

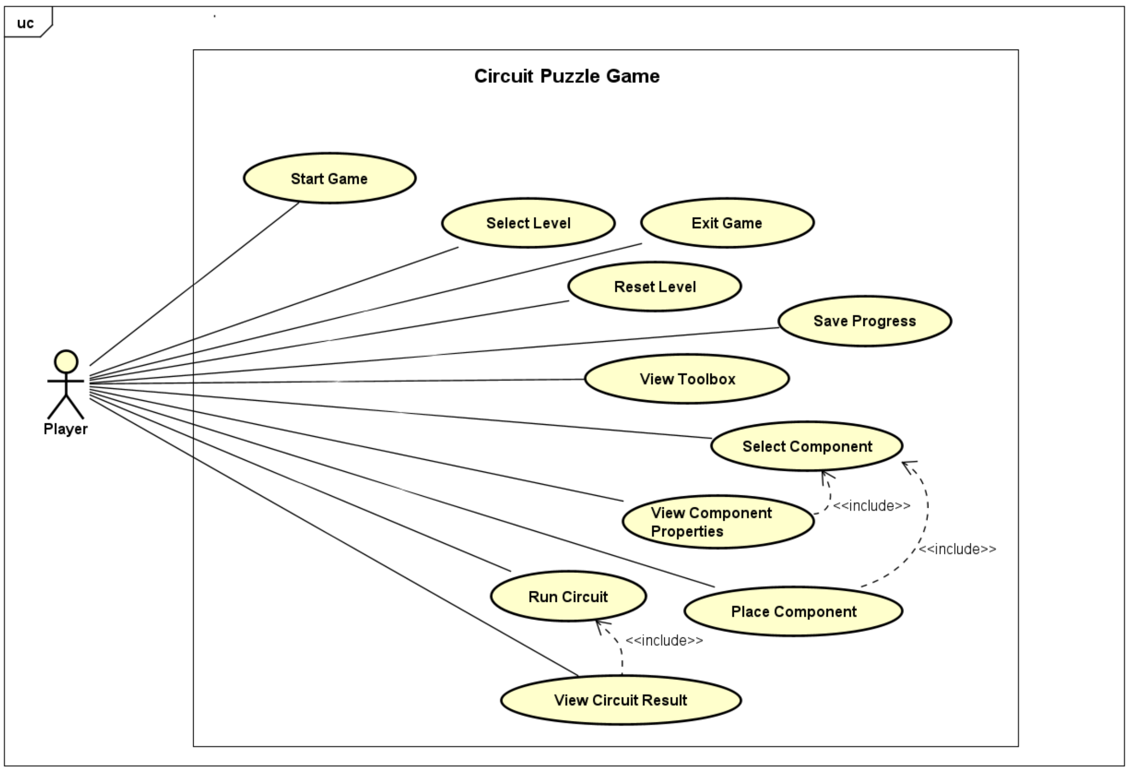

Figure 1: Use Case Diagram of Circuit Puzzle Game

### 2.2. Main Use Cases

- **Start Game**: Initialize the game session.
- **Select Level**: Choose Series or Parallel circuit.
- **View Toolbox**: See available components.
- **Select Component**: Choose a component to place.
- **Place Component**: Drag component onto the board.
- **View Component Properties**: Inspect resistance or capacitance.
- **Run Circuit**: Execute simulation.
- **View Circuit Result**: See calculated results.
- **Reset Level**: Clear board and retry.
- **Exit Game**: Close the application.

### 2.3. Use Case Relationships

- `Select Component` **includes** `View Component Properties`.
- `Run Circuit` **includes** `View Circuit Result`.
- `Play Tutorial` **extends** `Start Game`.

## III. Design

This section presents the detailed software design of the **Circuit Puzzle Game**. The design follows object-oriented principles and is organized into multiple logical packages and classes. Class diagrams and a use case diagram are used to illustrate the system structure, responsibilities of classes, and interactions between the player and the game.

### 1. General Class Diagram

The general class diagram presents a high-level view of the **Circuit Puzzle Game** architecture. At this level, class attributes and operations are intentionally omitted in order to focus on package structure and major conceptual relationships.

As illustrated in Figure 2, the system is divided into three main packages: **Component**, **Board**, and **Utils**.

The **Component** package contains all circuit components that can be placed on the board. The abstract base class `Component` is extended by concrete components such as `Wire`, `Resistor`, `Capacitor`, `Source`, `Destination`, `Bulb`, `Block`, `CornerWire`, and `TWire`. This design enables polymorphic handling of all circuit elements.

The **Board** package defines the circuit structure and evaluation rules. Different circuit configurations are represented by `SeriesBoard` and `ParallelBoard`, allowing the same components to be evaluated under different topologies.

The **Utils** package provides supporting utility classes such as `ClassUtils`, which contain helper functions used by both board logic and the user interface.

**Overall**, the general diagram gives us an overview of the structure of a game like Circuit Puzzle Game, allowing us to begin delving deeper into each package.

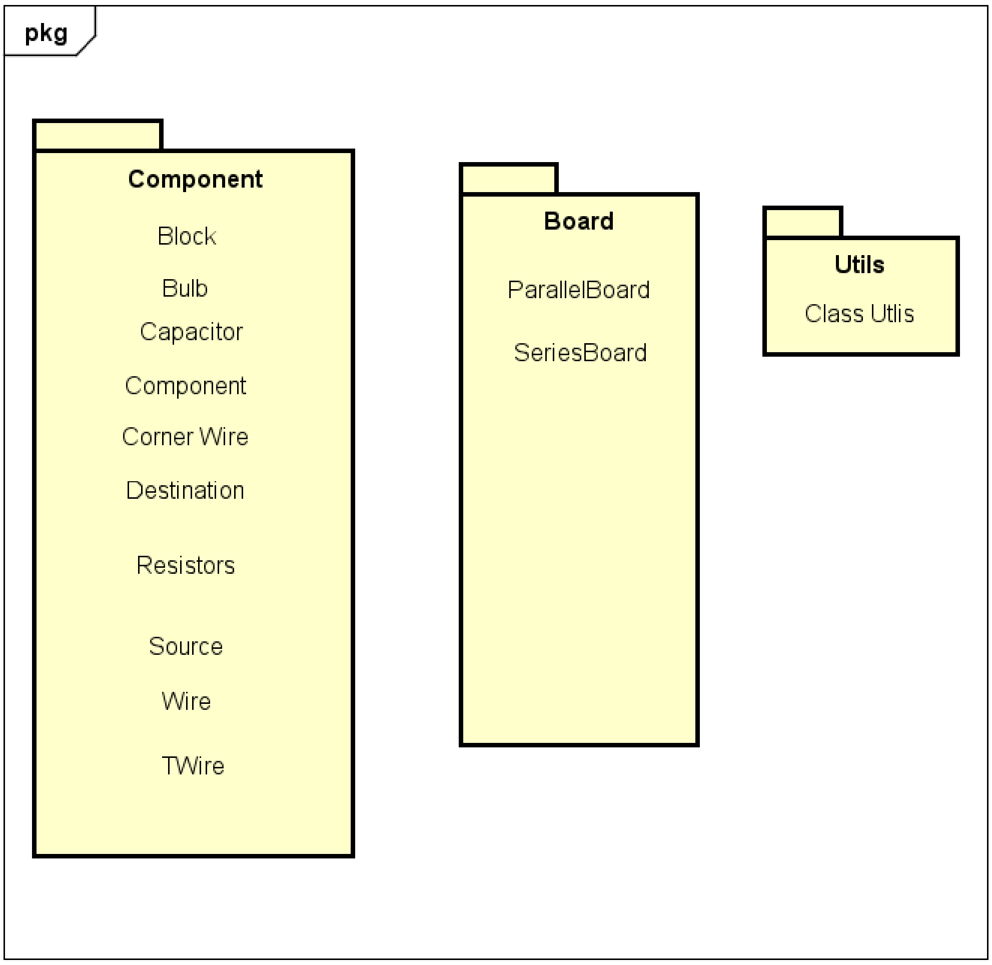

Figure 2: General Class Diagram of the Circuit Puzzle Game

## 2. Detailed Class Diagrams

This subsection describes each package in detail based on the provided class diagrams, including attributes, operations, and design intent.

### 2.1. Component Package Design

The **Component** package defines all circuit components that can appear on the board. The core of this package is the abstract base class `Component`.

**Component abstract class** The `Component` class represents a generic circuit element. It defines common attributes and behaviors shared by all components:

- **name**: Human-readable identifier of the component.
- **voltage**: Electrical voltage across the component.
- **current**: Electrical current flowing through the component.
- **isLocked**: Indicates whether the component is fixed and cannot be removed or rotated.
- **rotationDegree**: Stores the rotation state (0, 90, 180, 270 degrees), mainly used by wire components.

The class provides getter/setter methods and defines the abstract behavior `calculateAttributes()`, which is overridden by subclasses to compute their specific electrical properties.

**Concrete Component classes** All concrete components inherit from `Component`, enabling polymorphic behavior:

- **Wire**: A straight wire allowing electrical flow without resistance.
- **CornerWire**: A wire with a 90-degree turn, used to change current direction.
- **TWire**: A T-shaped wire enabling branching connections.
- **Resistor**: Stores a resistance value and affects total circuit resistance.
- **Capacitor**: Stores capacitance, previous voltage, and uses a fixed simulation time step for dynamic behavior.
- **Source**: Supplies voltage to the circuit.
- **Destination**: Represents the circuit endpoint.
- **Bulb**: Indicates circuit success; it lights up when current flows correctly.
- **Block**: Represents an obstacle cell that blocks placement and connections.

### 2.2. Board Package Design

The **Board** package manages the circuit grid and evaluation logic.

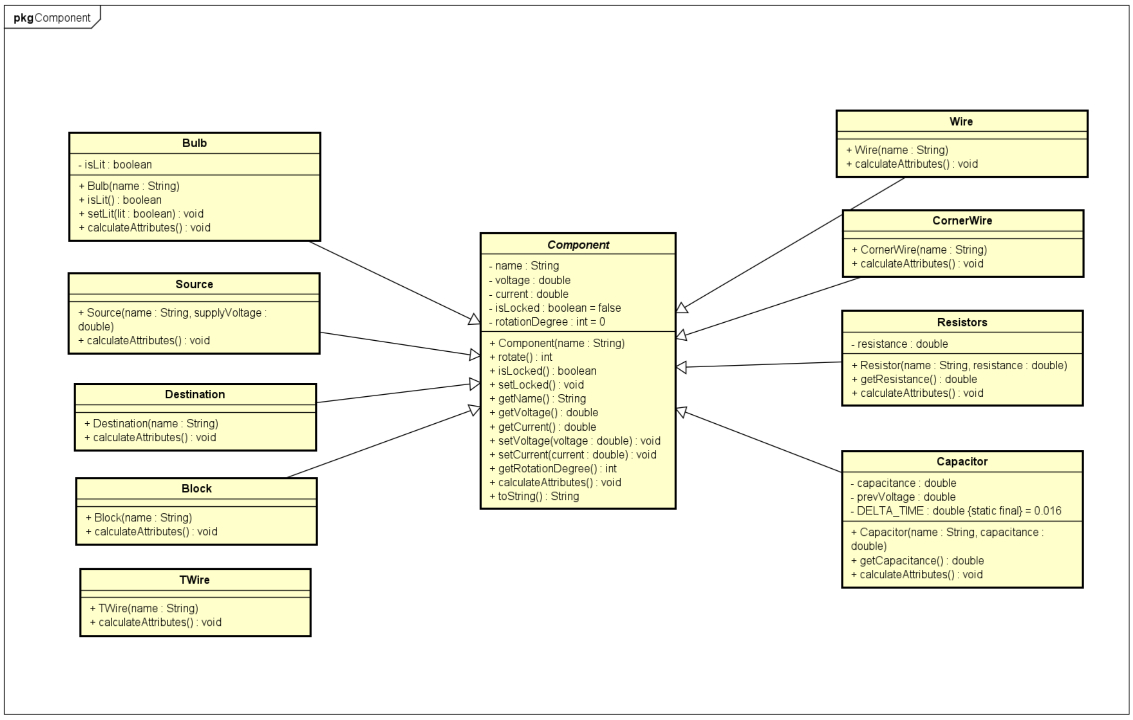
pkgComponent
Bulb
- isLit : boolean
+ Bulb(name : String)
+ isLit() : boolean
+ setLit(lit : boolean) : void
+ calculateAttributes() : void
Source
+ Source(name : String, supplyVoltage : double)
+ calculateAttributes() : void
Destination
+ Destination(name : String)
+ calculateAttributes() : void
Block
+ Block(name : String)
+ calculateAttributes() : void
TWire
+ TWire(name : String)
+ calculateAttributes() : void
Component
- name : String
- voltage : double
- current : double
- isLocked : boolean = false
- rotationDegree : int = 0
+ Component(name : String)
+ rotate() : int
+ isLocked() : boolean
+ setLocked() : void
+ getName() : String
+ getVoltage() : double
+ getCurrent() : double
+ setVoltage(voltage : double) : void
+ setCurrent(current : double) : void
+ getRotationDegree() : int
+ calculateAttributes() : void
+ toString() : String
Wire
+ Wire(name : String)
+ calculateAttributes() : void
CornerWire
+ CornerWire(name : String)
+ calculateAttributes() : void
Resistors
- resistance : double
+ Resistor(name : String, resistance : double)
+ getResistance() : double
+ calculateAttributes() : void
Capacitor
- capacitance : double
- prevVoltage : double
- DELTA_TIME : double {static final} = 0.016
+ Capacitor(name : String, capacitance : double)
+ getCapacitance() : double
+ calculateAttributes() : void

Figure 3: Detailed Class Diagram of Component

**CircuitBoard class:** `CircuitBoard` represents the main game board and acts as a container for all components. Its responsibilities include:

- `rows :int, cols :int` — board dimensions.
- `grid :Component[][]` — 2D matrix storing the placed components.
- `toolbox :List<Component>` — list representing available/selected components for placement (game inventory concept).
- `maxResistors :int, maxCapacitors :int` — global placement limits per level.

Key operations include:

- `presetComponent()` — initialize a puzzle (place fixed Source/Destination/Bulb/Block etc.).
- `placeComponent(r,c,comp) : boolean` — attempt to place a component into a cell.
- `removeComponent(r,c) : boolean` — remove a component if it is not locked.
- `getComponent(r,c) : Component` — retrieve a component at a cell.
- `clearGrid()` — reset the board state.

- `canAdd(type) : boolean` — verify component-limit constraints.
- `isValid(r1,c1,r2,c2) : boolean` — validate coordinate relation/bounds for adjacency checks.
- `canConnect(comp,rowDir,colDir) : boolean` — validate directional connectivity/ports of a component.
- `calculateTotalResistance() : double` — compute equivalent resistance.
- `calculateTotalCapacitance() : double` — compute equivalent capacitance.
- `getRows(), getCols()` — expose board dimension to the UI.

**SeriesBoard and ParallelBoard subclasses:** Two specialized boards inherit from `CircuitBoard`:

- **SeriesBoard**: Implements resistance and capacitance calculations for series circuits.
- **ParallelBoard**: Implements resistance and capacitance calculations for parallel circuits.

Each subclass overrides calculation methods and the `presetComponent()` method to initialize predefined layouts and locked components.

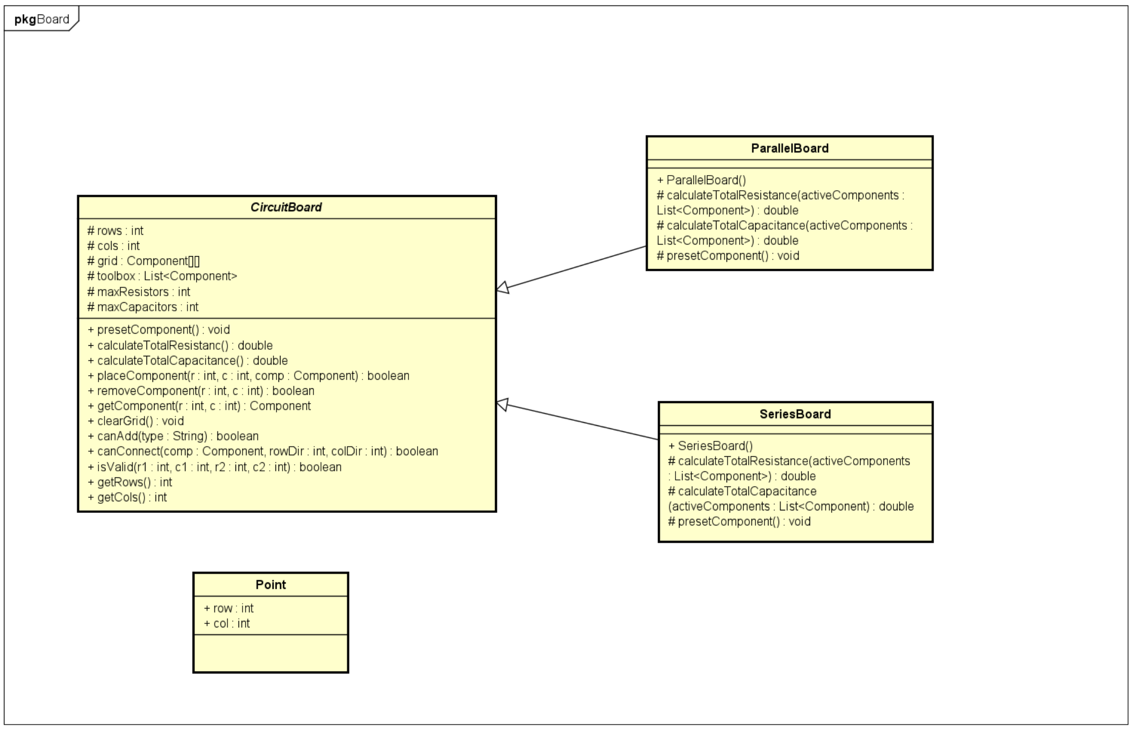
pkgBoard
CircuitBoard
# rows : int
# cols : int
# grid : Component[][]
# toolbox : List<Component>
# maxResistors : int
# maxCapacitors : int
+ presetComponent() : void
+ calculateTotalResistanc() : double
+ calculateTotalCapacitance() : double
+ placeComponent(r : int, c : int, comp : Component) : boolean
+ removeComponent(r : int, c : int) : boolean
+ getComponent(r : int, c : int) : Component
+ clearGrid() : void
+ canAdd(type : String) : boolean
+ canConnect(comp : Component, rowDir : int, colDir : int) : boolean
+ isValid(r1 : int, c1 : int, r2 : int, c2 : int) : boolean
+ getRows() : int
+ getCols() : int
ParallelBoard
+ ParallelBoard()
# calculateTotalResistance(activeComponents : List<Component>) : double
# calculateTotalCapacitance(activeComponents : List<Component>) : double
# presetComponent() : void
SeriesBoard
+ SeriesBoard()
# calculateTotalResistance(activeComponents : List<Component>) : double
# calculateTotalCapacitance (activeComponents : List<Component) : double
# presetComponent() : void
Point
+ row : int
+ col : int

Figure 4: Detailed Class Diagram of Board

### 2.3. Utils Package Design

The **Utils** package contains stateless helper classes.

**ConnectionLogic class:** This utility class provides static methods to analyze circuit connectivity:

- `getActivePorts(Component)`: Determines which ports of a component are active.
- `areConnected(source,target,direction)`: Checks whether two components are connected.
- `getFlowCount(board,component)`: Calculates current flow paths through a component.

**GuiUtils class:** `GuiUtils` supports graphical rendering and visual indicators. For example, it adds connection markers to the JavaFX UI to visualize valid links between components.

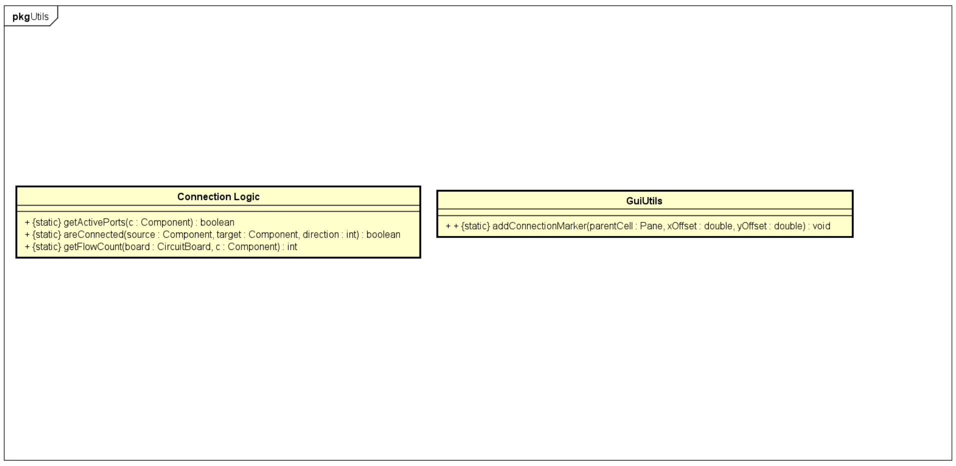
pkgUtils
Connection Logic
+ {static} getActivePorts(c : Component) : boolean
+ {static} areConnected(source : Component, target : Component, direction : int) : boolean
+ {static} getFlowCount(board : CircuitBoard, c : Component) : int
GuiUtils
+ + {static} addConnectionMarker(parentCell : Pane, xOffset : double, yOffset : double) : void

Figure 5: Class Diagram of Utils

## 3. Explanation of Design Relationships

### 3.1. Inheritance (Generalization)

Inheritance is widely used to promote code reuse and extensibility:

- All circuit elements inherit from `Component`.
- `SeriesBoard` and `ParallelBoard` inherit from `CircuitBoard`.

This allows the board to operate on generic `Component` references while preserving component-specific behavior at runtime.

### 3.2. Composition

`CircuitBoard` owns a 2D array of `Component` objects. This represents a strong composition relationship: components exist only within the context of a board.

### 3.3. Dependency

The main game class depends on:

- `CircuitBoard` for game logic,
- `Component` subclasses for gameplay interaction,
- `ConnectionLogic` for validation,
- `GuiUtils` for rendering support.

## 4. Implementation of Important Methods

### 4.1. Component Placement

The `placeComponent()` method ensures:

- Valid grid position,
- Target cell is not locked,
- Component limits are respected,
- Connectivity rules remain valid.

### 4.2. Connectivity Checking

Connectivity is verified using a graph traversal approach (BFS/DFS) across the grid, ensuring a continuous path from `Source` to `Bulb` or `Destination` through valid components.

### 4.3. Circuit Evaluation

Total resistance and capacitance are calculated based on board type:

$$R_{\text{series}} = \sum R_i, \quad \frac{1}{R_{\text{parallel}}} = \sum \frac{1}{R_i}$$

$$C_{\text{series}}^{-1} = \sum C_i^{-1}, \quad C_{\text{parallel}} = \sum C_i$$

The bulb lighting duration is derived from the RC time constant:

$$\tau = R \times C$$