PCB-GUARD: A PROGRAM FOR PCB DESIGN WITH LAYOUT VALIDATOR

Tran Phuoc Nguyen Khoi-102240121[a,1], Dang Chi Kien-102240010[a,1], Huynh Phuoc Vinh-102240487[a,1]

[a]*Department of Electrical and Computer Engineering, Vietnamese-German University, Ring Road 4, Thoi Hoa, Ho Chi Minh City, 75911, Vietnam*

Abstract

In the Electrical and Computer Engineering (ECE) Program, students have a chance to study about printed circuit boards (PCBs) in the Engineering Design module. And PCB design is a fundamental problem that requires high precision. As circuit complexity increases, the risk of human error during the manual placement of components - such as overlapping footprints, physical interference, or violation of clearance constraints - rises significantly. These errors can lead to multiple circuitry failures and unexpected costs. The specific problem addressed in this project is the automated validation of component spatial arrangement. The goal is to develop a program that simulates the PCB design process and detects spatial conflicts. To solve this problem, the project employs a C++ Object - Oriented Programming (OOP) approach combined with Electrical Engineering Design principle. A Validator algorithm will be deployed for checking and detecting errors in the design process. Upon completion, the project aims to deliver a functional C++ console application with following capabilities: PCB environment simulation and validator feature.

1. Introduction

PCB-GUARD: A Program for PCB Design with Layout Validator

Popular PCB design applications such as Autodesk EAGLE, AutoCAD Electrical, Altium Designer, and KiCad provide comprehensive environments for schematic capture, component placement, and PCB layout design. These tools typically follow a schematic-driven workflow, where the logical circuit design is translated into a physical PCB layout.

Printed circuit board (PCB) design is a core topic in the Electrical and Computer Engineering (ECE) curriculum and requires a high level of precision. As circuit complexity increases, manual component placement becomes more error-prone, leading to issues such as component overlap and clearance violations.

To ensure design correctness, commercial PCB software applies rule-based validation, commonly known as Design Rule Check (DRC). This method verifies the layout against such as component footprint overlap or board boundary limits.

Despite their effectiveness, these tools present several limitations. It is harsh to limit the distance between components to place them on the circuit board or about the licensing cost and hardware requirements can restrict accessibility.

In this project, we introduce a C++ based PCB design simulation tool automatically validates component placement using object-oriented programming and electrical engineering design principles, while eliminating obstacles such as licensing cost, hardware requirements or other restrictions.

Features of the system:

- For those interested in circuit board building.
- Circuit board design.
- Report minor errors.
- Create Users and Products record.
- Delete Users and Products record.
- Show specific Users and Products record.
- Show all Users and Products record.

2. Methodology

2.1. Problem description

In designing PCB, a PCB board serve as a working environment, where electrical components will be place on the PCB board, which are then connected by trace connections. However, we need to take into account in geometry navigation. Our mission is to create electrical components, and evaluate the position of each components on the PCB by using mathematical equation. To achieve this, the system is specified into four classes: a base class Component, a class template Grid for getting coordinate, a class Trace for connection, and a class PCB to manage the overall state.

2.2. Coordinate System

The Grid class template is the core for the fundamental workspace of the PCB. By creating a dynamically allocated $2D$ array (matrix), it can store the state of every point on the board. The coordinate system operates on a grid of dimensions $W \times H$. Each cell is accessed via (x, y) coordinates, where x and y represent the the column and the row index, respectively. The class manages the memory by allocating and deallocating, which is ensuring the memory safety by validating all coordinate accesses against the defined board boundaries.

Email addresses: 102240121@student.vgu.edu.vn (Tran Phuoc Nguyen Khoi-102240121), 102240010@student.vgu.edu.vn (Dang Chi Kien-102240010), 102240487@student.vgu.edu.vn (Huynh Phuoc Vinh-102240487)

[1]These authors contributed equally to this work.

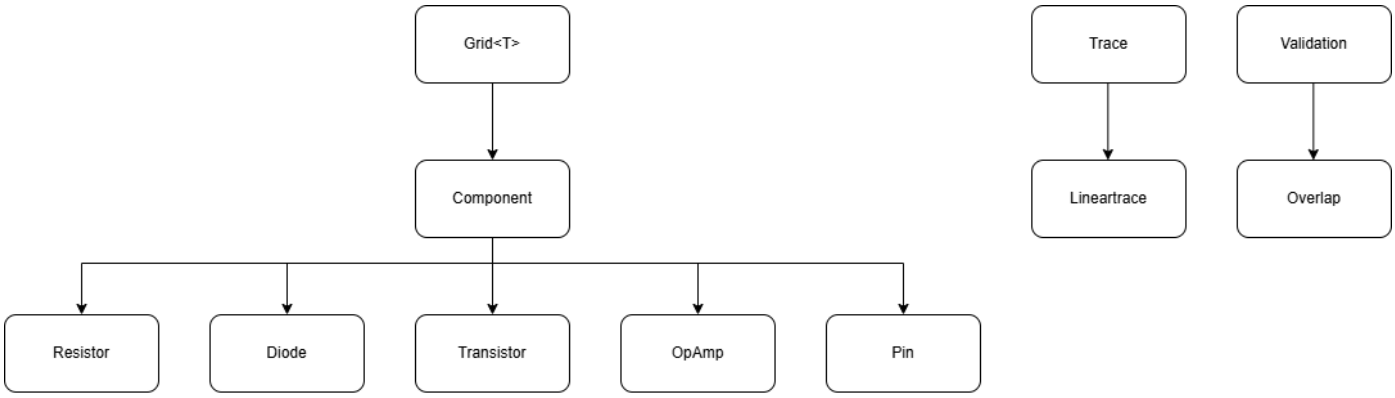


Figure 1: Class Hierarchy

2.3. Components and Overlap Evaluation

2.3.1. Component Abstraction

Here, we assume there are only six components which are Resistors, Opamp, Transistor, Diode, Header pin with 2 ports and 3 ports inherit from a base class Component. In order to define the geometry of each element, we let $P(x, y)$ represent center coordinate together with a bounded rectangular region in the $2D$ grid $\mathbb{Z}^2$, which assigns the width L and height H based on the component type.

This geometric definition provides the spatial data required for subsequent collision detection algorithms.

2.3.2. Overlap Detection Algorithm

One of the important requirements for PCB design is preventing physical collisions. By creating the Boundary Box of each components (**Figure 2.**), we can evaluate the position of every component against others. In order to evaluate, we first denote the center position of component A is (x_A, y_A) and component B is (x_B, y_B). Then the collision is defined by checking for the absence of a separating gap by computing the coordinate of the bounding box, which are the Left, Right, Bottom, and Top of each element. For component A, we have the equation:

$$L_A = x_A - \frac{L}{2}, \quad R_A = x_A + \frac{L}{2} \tag{1}$$

$$Top_A = y_A - \frac{W}{2}, \quad Bottom_A = y_A + \frac{W}{2} \tag{2}$$

Two components A and B are overlapping if and only if none of the following conditions are true.

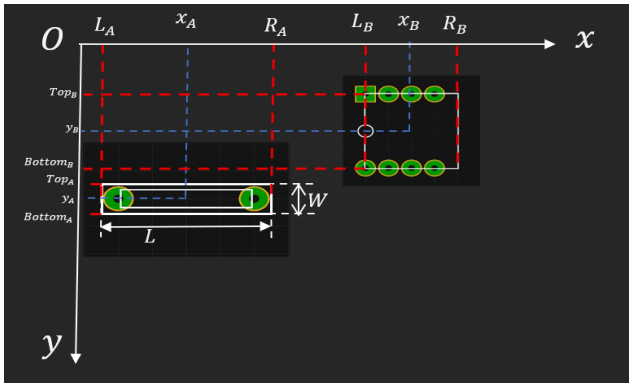


Figure 2: The Overlap detection algorithm.

- A is strictly to the left of B ($R_A < L_B$).
- A is strictly to the right of B ($L_A > R_B$).
- A is strictly above B ($Bottom_A < Top_B$).
- A is strictly below B ($Top_A > Bottom_B$).

The overall algorithm is shown at **Algorithm [1]**.

Algorithm 1: Check Overlap (Bounding Box)

Input: $A(x_A, y_A, w_A, h_A)$, $B(x_B, y_B, w_B, h_B)$
$L_A \leftarrow x_A - w_A/2$; $R_A \leftarrow x_A + w_A/2$;
$L_B \leftarrow x_B - w_B/2$; $R_B \leftarrow x_B + w_B/2$;
$Top_A \leftarrow y_A - h_A/2$; $Bottom_A \leftarrow y_A + h_A/2$;
$Top_B \leftarrow y_B - h_B/2$; $Bottom_B \leftarrow y_B + h_B/2$;
if $R_A < L_B$ ***or*** $L_A > R_B$ ***or*** $Bottom_A < Top_B$ ***or***
$Top_A > Bottom_B$ **then**
  return False;
return True;

Beside physical collision, it is crucial to have a minimum distance between components. This is because it will have electrical interference or soldering error, when two components are placed near to each other. For solving this problem, we develop ViolateClearance algorithm with additional minimum clearance value. Its basic principle is based on the Bounding

Box (BB) algorithm, but it will extend the bounding box outwards by the clearance value. The overall can be illustrated in **Algorithm.[2]**.

Algorithm 2: Check ViolateClearance

Input: $A(x_A, y_A, w_A, h_A)$, $B(x_B, y_B, w_B, h_B)$, Minimum Clearance M
Output: True if violate the clearance
$L_A \leftarrow x_A - w_A/2 - M$; $R_A \leftarrow x_A + w_A/2 + M$;
$L_B \leftarrow x_B - w_B/2$; $R_B \leftarrow x_B + w_B/2$;
$Top_A \leftarrow y_A - h_A/2 - M$; $Bottom_A \leftarrow y_A + h_A/2 + M$;
$Top_B \leftarrow B.y - B.h/2$; $Bottom_B \leftarrow B.y + B.h/2$;
if $R_A < L_B$ ***or*** $L_A > R_B$ ***or*** $Bottom_A < Top_B$ ***or*** $Top_A > Bottom_B$ **then**
  return False;
return True;

2.4. Linear System Working Principle

2.4.1. Trace Connection and Bresenham's Algorithm

Unlike components, which has simple geometry, traces that are used to connect part on PCB are having complex geometry, which will have 90 degree or 45 degree turn corner. To address this, we split the multi-turn line into small individual straight line segments. To visualize these points, we use Bresenham's Algorithm. The basic principle of this algorithm is to determine which cell on the grid should be marked.

To do so, we are denoting P is set of points $\{p_0, p_1, \ldots, p_{N-1}\}$ and each point will contain coordinate (x_{pm}, y_{pm}). Then with a pair of points, which are $p_m = (x_m, y_m)$ and $p_{m+1} = (x_{m+1}, y_{m+1})$, we implement Bresenham's Line Algorithm to map continuous segments onto PCB grid. We compute the absolute distance and step directions on the x and y axis, respectively.

$$\Delta x = |x_{m+1} - x_m|, \quad s_x = \text{sgn}(x_{m+1} - x_m) \tag{3}$$
$$\Delta y = -|y_{m+1} - y_m|, \quad s_y = \text{sgn}(y_{m+1} - y_m) \tag{4}$$

Where

$$s_x = \begin{cases} 1 & \text{if } x_{m+1} > x_m \\ -1 & \text{otherwise} \end{cases} \tag{5}$$

and

$$s_y = \begin{cases} 1 & \text{if } y_{m+1} > y_m \\ -1 & \text{otherwise} \end{cases} \tag{6}$$

An error term e is introduced to track the deviation of the pixel path from the true geometric line. It is defined as $e_m = \Delta x + \Delta y$. The coordinates for the next step (x_{m+1}, y_{m+1}) are derived iteratively based on the current error e_m:

$$x_m = \begin{cases} x_m + s_x & \text{if } 2e_m \geq \Delta y \\ x_m & \text{otherwise} \end{cases} \tag{7}$$

$$y_m = \begin{cases} y_m + s_y & \text{if } 2e_m \leq \Delta x \\ y_m & \text{otherwise} \end{cases} \tag{8}$$

This method ensures that the trace is drawn using only integer calculations, matching the internal logic of the simulation grid. To illustrate this, **algorithm.[3]** is the process of breaking complex paths into simpler ones. Its basic principle is that we consider a complex path is composed of many straight line segments and with each line we apply the **algorithm.[4]** to show the individual line.

Algorithm 3: Polyline Trace Processing

Input: Array of Points $P = \{p_0, p_1, \ldots, p_{N-1}\}$, N: number of point, $p_i = (x_i, y_i)$
for $i \leftarrow 0$ **to** $N - 1$ **do**
  $p_i \leftarrow P[i]$;
  $p_{i+1} \leftarrow P[i + 1]$;
  Bresenham's Algorithm(())

Algorithm 4: Bresenham's Line Algorithm (Integer Arithmetic)

Input: (x_m, y_m),(x_{m+1}, y_{m+1})
$\Delta x \leftarrow |x_{m+1} - x_m|$;
if $x_m < x_{m+1}$ **then**
  $s_x = 1$
$\Delta y \leftarrow -|y_1 - y_0|$;
if $y_m < y_{m+1}$ **then**
  $s_y = -1$
$err \leftarrow \Delta x + \Delta y$;
while *true* **do**
  if $0 \leq x_m < Width$ ***and*** $0 \leq y_m < Height$ **then**
    $G(x_m, y_m) \leftarrow$ "+" ; `// mark the plus sign on the Grid`
  if $x_m = x_{m+1}$ ***and*** $y_m = y_{m+1}$ **then**
    break;
  $e_2 \leftarrow 2 \times err$;
  if $e_2 \geq \Delta y$ **then**
    $err \leftarrow err + \Delta y$;
    $x_m \leftarrow x_m + s_x$;
  if $e_2 \leq \Delta x$ **then**
    $err \leftarrow err + \Delta x$;
    $y_m \leftarrow y_m + s_y$;

3. Code implementation

3.1. Core Coordinate System

we define a class template Grid in header file *Grid.h*, with variable width and height, which is part of 2D array. This class provide the core 2D coordinate for the program.

3.2. Component and Validation

3.2.1. Component definition

First, the algorithm for component abstraction is first use for defining each component. By declaring constructor Component:

```
Component(const int&id_, const int&x_,
    const int&y_, const int&width_, const
    int&height_, const string& sym_);
```

Class Resistor, Diode, Transistor, Opamp, and Pin

The class of each element will inherit from the base implementation which is the Base Class *Component* and the declare function:

```
virtual void
    Component::drawonGrid(Grid<string>&
    comp);
```

This function is to visualize the component on the grid and it is also overridden as it will have different draw function to draw because of different shape.

3.2.2. Overlap and Clearance validation

After defining the component, the component's absolute position is extracted to conduct the boundary and ViolateClearance check which follow the **Algorithm.[1] and Algorithm.[2]** to check if the component is placed outside of the working place or whether it is violate the clearance between two components. The algorithm function is declare in **Component.h**:

Component.h

```
bool Component::overlaps(const Component&
    other) const;

bool Component::violatesClearance(const
    Component& other, int minClearance)
    const;
```

3.3. Linear Trace

The routing logic is encapsulated in the `Lineartrace` class in *Lineartrace.cpp*. This implementation bridges the high-level data structure of a wire with the low-level pixel manipulation required for the grid.

```
void drawonGrid(Grid<std::string>& grid)
    const override;
```

This function is declared which contain the Polyline Trace Processing and Bresenham's Line algorithm, which is **algorithm.[3] and algorithm.[4]**, respectively.

3.4. Visualization and Control (SFML)

Our advance feature is to utilize the Graphics module of the Simple and Fast Multimedia Library (SFML). The installation step is consist of:

- Downloading MSYS2.
- From MSYS2 terminal, download the library by using command $pacman - S mingw - w64 - x86_64 - sfml$.
- Once installed, we can include the library in the beginning of the main file and run normally.

All code that are related to new library is used in *PCBViewer.h*. This library is used only to create graphic for trace and components for easy to visualize and evaluate.

4. Results and Discussion:

4.1. Results

For getting the user input, we defining two inputs files which is *Component.txt* and *Traces.txt* (**Figure.[3]** and **Figure.[4]**)

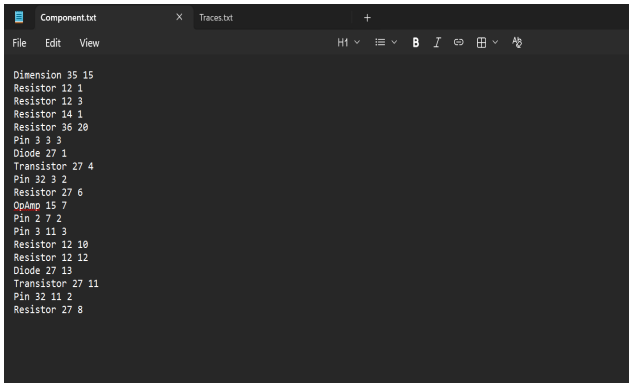


Figure 3: Component User Input

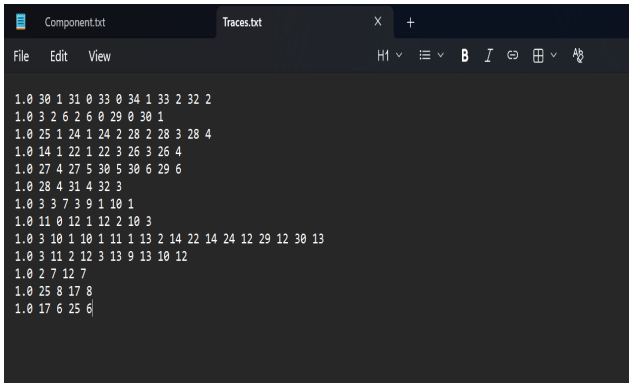


Figure 4: Component User Input

4.1.1. *Components on the circuit board*

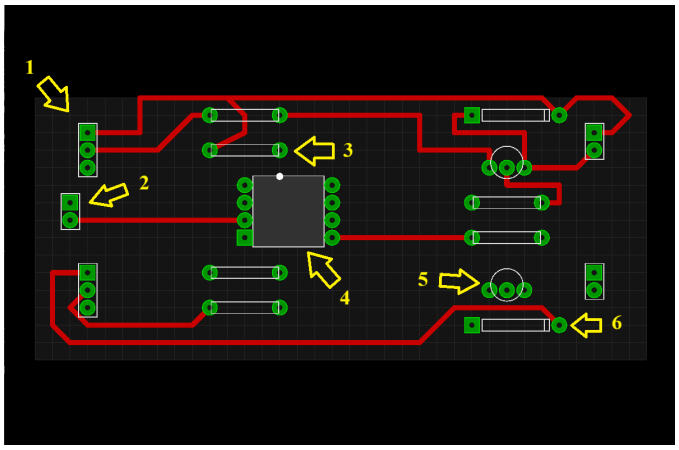


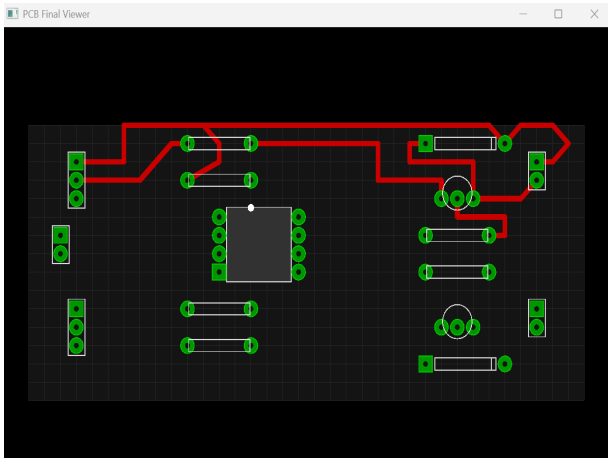
Figure 5: PCB Component Layout

1. Pin Header (2 holes)
2. Pin Header (3 holes)
3. Resistor
4. IC Opamp
5. Transistor
6. Diode

4.2. *Discussion*

Here, we will discuss the case when we do PCB design:

- Case 1: No error.
- Case 2: Component Overlap each other (No Clearance checking).
- Case 3: Component are violated the Clearance which is the minimum space between the component (No Overlap).



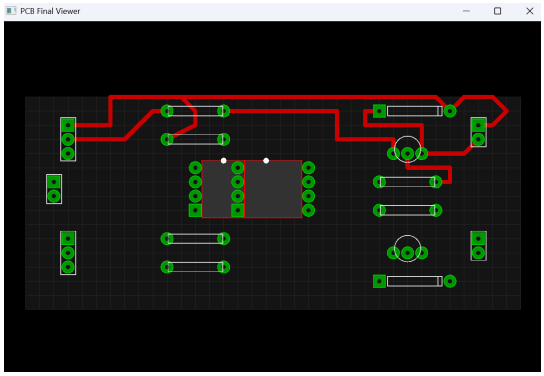
(a) Visualize in Graphic

```
--- Initializing Components ---
--- Validating Board ---
Starting PCB Validation Check
NO ERRORS FOUND

--- PCB Grid View ---
   01234567890123456789012345678901234
0 |......++++++++++++++++++++++++.+++.|
1 |......+..+oR1Ro++++++++.+oD1D|o...+|
2 |...o+++.+...+.........+.+++++...o+.|
3 |...o++++..oR2Ro.......++++TTT...o..|
4 |...o.........OO^O.........ooo+++...|
5 |............oOOOOo.........TTT++....|
6 |..o.........oOOOOo.......oR3Ro+....|
7 |..o.........oOO1Oo..................|
8 |............oOOOOo.......oR6Ro.....|
9 |............OOOO....................|
0 |...o.......oR4Ro............TTT...o..|
1 |...o.........................ooo...o..|
2 |...o.......oR5Ro............TTT......|
3 |.........................oD2D|o....|
4 |.....................................|

Opening Graphical Viewer...
```

(b) In Console

Figure 6: Case 1: PCB Circuit with no error.

In Case 1 (**Figure.[6]**), we assume there is no requirement in minimum clearance between the component. Therefore the result will not have any errors as shown in Figure.[6b]. Then, we evaluate the next case by adding another OPAMP such that it will overlap the other OPAMP (Figure.[7]).



(a) Visualize in Graphic

```
--- Initializing Components ---
--- Validating Board ---
Starting PCB Validation Check
ERROR: Overlap detected between O(ID:1) and O(ID:2)
Violation: Overlap detected between O(ID:1) and O(ID:2)
ERRORS DETECTED

--- PCB Grid View ---
   01234567890123456789012345678901234
0 |......++++++++++++++++++++++++.+++.|
1 |......+..+oR1Ro++++++++.+oD1D|o...+|
2 |...o+++.+...+.........+.+++++...o+.|
3 |...o++++..oR2Ro.......++++TTT...o..|
4 |...o.........OO^OO^O......ooo+++...|
5 |............oOOoOOOOo.....TTT++....|
6 |..o.........oOOoOOOOo....oR3Ro+....|
7 |..o.........oOOoOO2Oo..............|
8 |............oOOoOOOOo....oR6Ro.....|
9 |.............OOOOOOO...............|
0 |...o......oR4Ro...........TTT...o..|
1 |...o.......................ooo...o..|
2 |...o......oR5Ro...........TTT......|
3 |.........................oD2D|o....|
4 |.................................|
```

(b) In Console

Figure 7: Case 2: Checking Overlapping Components

In Case 2, the console will show the result which is "Overlap detected between O(ID:1) and O(ID:2)" and in graphic visualization, it will mark the overlap element with red boundary to indicate the error for user in **Figure.[7b]** and **Figure.[7a]**, respectively.

```
--- Initializing Components ---
--- Validating Board ---
Starting PCB Validation Check
ERROR: Clearance violation between R(ID:2) and O(ID:1)
ERROR: Clearance violation between T(ID:1) and R(ID:3)
ERROR: Clearance violation between R(ID:4) and O(ID:1)
ERROR: Clearance violation between D(ID:2) and T(ID:2)
Violation: Clearance violation between D(ID:2) and T(ID:2)
ERRORS DETECTED

--- PCB Grid View ---
   01234567890123456789012345678901234
0 |......++++++++++++++++++++++++.+++.|
1 |......+..+oR1Ro++++++++.+oD1D|o...+|
2 |...o+++.+...+.........+.+++++...o+.|
3 |...o++++..oR2Ro.......++++TTT...o..|
4 |...o.........OO^O.........ooo+++...|
5 |............oOOOOo........TTT++....|
6 |..o.........oOOOOo.......oR3Ro+....|
7 |..o.........oOOlOo..................|
8 |............oOOOOo.......oR6Ro.....|
9 |.............OOOO...................|
0 |...o......oR4Ro...........TTT...o..|
1 |...o.......................ooo...o..|
2 |...o......oR5Ro...........TTT......|
3 |.........................oD2D|o....|
4 |.................................|
```

Figure 8: Violation Checking with ViolateClearance requirement.

As regard for Case 3, we take into account the minimum violation clearance which is a constant value of minimum distance between two components ($k = 1$). The console will print out to the console the list of components are violated, so the user can detect and fix the design, which is as shown in **Figure.[8]**.

In conclusion, our PCB validator has successfully evaluate all the cases that can represent human error in designing PCB.

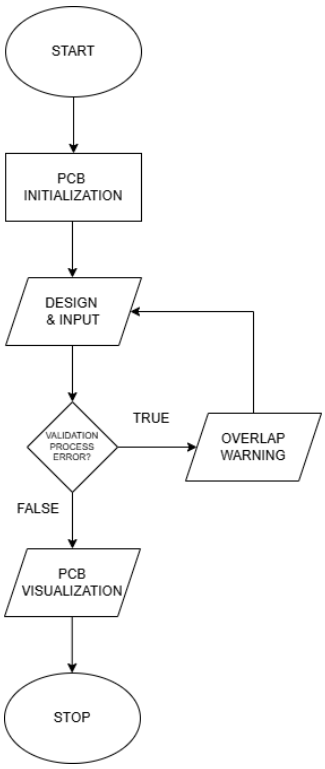


Figure 9: PCB-Guard's flowchart

5. External and Internal Libraries

5.1. External Library: SFML

SFML (Simple and Fast Multimedia Library) is a cross-platform API designed to provide a simple interface to various components of the PC. In this project, SFML is utilized specifically to replace the console output with a graphical PCB view.

A key strength of this library lies in its ability to capture user inputs (mouse, keyboard) and render $2D$ shapes in real-time.

5.1.1. How to use SFML

- To initialize the application window, use the constructor of `sf::RenderWindow`:

```
sf::RenderWindow
    window(sf::VideoMode(800, 600),
    "PCB GUARD");
```

- To process the main loop and handle events (like closing the window), invoke the `pollEvent()` function:

```
while (window.pollEvent(event)) \{ ...
    \}
```

- To draw a specific component (shape) onto the buffer, call the following function:

```
window.draw(drawableObject);
```

- To display the rendered frame on the screen (Duration is managed by the frame rate), use the function:

```
window.display();
```

- To stop the application and release the window resources, use these functions:

```
{window.close();
```

5.2. *Internal Standard Library*

As necessary for data management and mathematical operations, the following standard libraries are included:

Function or Library	**Meaning**
<iostream>	Used for input/output stream (console debugging).
<vector>	Arrays in this program are declared in fixed sizes. However, storing arbitrary amounts of input data (components added by user) requires flexibility. The `vector` library provides dynamically-sized arrays.
<cmath>	Mathematical functions (sqrt, pow) for distance check. Especially for the distance calculation in the `distanceTo()` function, they work with Euclidean geometry.
<memory>	This library supports handling memory management automatically. To declare a unique pointer, use the syntax: `std::unique_ptr<Type>`.
`sf::Event`	Handles system events like closing window or key press.
<algorithm>	Provides a collection of functions for operations on ranges of elements. In our Validator system, we specifically use `std::min` and `std::max` to determine the boundaries of the Overlap Area (AABB logic).
`window.pollEvent()`	Pops the event on top of the event queue.

Table 1: Internal libraries and key functions

- <fstream>: This library supports handling files input.
- <sstream>: This library supports handling a text string.
- <map>: This library supports handling a collection of pairs to manage and count the number of components by type.

6. Conclusion

6.1. *Conclusion*

In this project, we successfully built a basic PCB Design Simulation tool using C++ and the SFML library. The application meets the initial requirements set by the team:

- A functional Grid System that allows users to snap components to specific coordinates.
- Implementation of Object-Oriented Programming (OOP) with a base `Component` class and derived classes for Resistors and ICs.
- Direct integration of Bresenham's Algorithm for drawing linear traces, ensuring efficient rendering on a pixel-based grid.
- A working Validator system using AABB (Axis-Aligned Bounding Box) logic to detect and prevent component overlaps in real-time.

Through the development process, the team gained practical experience in memory management (using `std::vector` and smart pointers) and graphical logic. The separation of the Logic Layer (Data) and View Layer (SFML) helped in debugging and maintaining the code structure.

6.2. *Self-evaluation*

Challenges:

- After coming up with the idea, the team faced difficulties in storing information for future purpose, and initial solution considered by the team was file handling.
- The logic of the methods and functions needed continuous adjustment to achieve the desired result.

Achievement:

- We dedicated a significant amount of time to the planning stage, which helped us discover new libraries and features we hadn't used before. Beyond just learning new tools, this project allowed us to solidify our C++ knowledge and practice turning theoretical ideas into actual running code. Most importantly, we realized the true value of OOP in organizing and managing a project of this size.

7. Limitations and Future Improvements

7.1. Current Limitations

To be honest, while the grid logic works, the application is currently more of a "viewer" than a fully functional CAD tool. We encountered several specific constraints during development:

- **Hardcoded Configuration:** This is the biggest drawback. Currently, the User Need a reference of PCB Grid to add and change the components and traces at runtime. If we want to change the circuit design (e.g., move a resistor), we have to manually edit the coordinate numbers.
- **Arbitrary Dimensions (No SI Units):** Real PCB components follow strict standards (footprints like 0805, DIP-8) measured in millimeters or mils. In our project, we simply "assumed" the sizes based on pixels (e.g., a resistor is 50×10 pixels). We did not apply the Metric System (SI), so the scale is not accurate for manufacturing.
- **Single-Layer Design:** A real PCB consists of multiple layers (Top Copper, Bottom Copper, Silk Layer, etc.). Our simulation operates strictly on a single $2D$ plane.
- **No Electrical Intelligence:** The system only detects "physical" collision (overlapping pixels). It lacks an electrical rule check, meaning it cannot identify a "Short Circuit" if a user accidentally connects VCC to GND.

7.2. Future Work

Based on the difficulties we faced, the next steps for this project would focus on usability:

- **Interactive Input System:** We need to build a menu where users can select a component and place it with the mouse. This will remove the need to modify the source text files for every design change.
- **Standard Component Library:** Instead of drawing generic rectangles, we should implement a library of real-world component dimensions (applying a Pixel-to-Millimeter conversion ratio).
- **Electrical Logic:** Implement a netlist system to understand connections, allowing the software to warn users about short circuits or broken nets.

8. Role and contribution of each member

Incharge	Tasks
All	Choose Topic: PCB Design Simulation.
All	Brainstorming features: Validation algorithm, Component classes, and SFML visualization.
Khôi	Research Grid data structure and coordinate system management ($2D$ Array).
Khôi	Develop the base `Component` class and define inheritance for specific parts (Resistor, IC...).
Vinh	Research Bresenham's Algorithm for drawing linear traces on a pixel grid.
Vinh	Implement `Trace` and `LinearTrace` classes to handle wire connections.
Kiên	Research SFML library for graphical user interface and visualization.
Kiên	Implement specific component drawing logic (Resistor, OpAmp, etc.) visuals.
Khôi	Write code for Overlap Detection and Clearance Validation algorithms (Bounding Box).
Khôi, Vinh	Integrate Components and Traces into the main PCB board controller.
Kiên	Write Introduction and Problem Description sections in the report.
Khôi	Write Methodology section: Coordinate System, Components, and Overlap Evaluation.
Vinh	Write Methodology section: Linear System, Trace Connection, and Algorithms.
All	Perform code debugging and test the Validator feature with sample scenarios.
All	Final review of the report and code refactoring before submission.

Table 2: Detailed Work Allocation

References

[1] Bresenham, J. E. (1965). Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1), 25-30. DOI: 10.1147/sj.41.0025

[2] Ericson, C. (2004). *Real-Time Collision Detection*. Morgan Kaufmann. (Chapter 4: Bounding Volumes).

[3] Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley Professional.

[4] Gomila, J. (2015). *SFML Game Development*. Packt Publishing Ltd.

[5] Coombs, C. F. (2007). *Printed Circuits Handbook* (6th ed.). McGraw-Hill Education.

[6] https://youtu.be/vLnPwxZdW4Y?si=zphnbVCaDcqDBoE5

Annex A. Grid.h

```
#ifndef GRID_H
#define GRID_H

#include <iostream>
#include <string>

template<typename T>
class Grid {
private:
    int width;
    int height;
    T** entries;

public:

    Grid(int w, int h) : width(w), height(h) {
        entries = new T*[height];
        for (int i = 0; i < height; ++i) {
            entries[i] = new T[width];
        }
    }


    ~Grid() {
        if (entries) {
            for (int i = 0; i < height; ++i) {
                delete[] entries[i];
            }
            delete[] entries;
        }
    }


    T& operator()(int x, int y) {
        if(x >= 0 && x < width && y >= 0 && y < height) {
            return entries[y][x];
        }
        static T dummy;
        return dummy;
    }


    const T& operator()(int x, int y) const {
        if(x >= 0 && x < width && y >= 0 && y < height) {
            return entries[y][x];
        }
        static T dummy;
        return dummy;
    }


    int getWidth() const { return width; }
    int getHeight() const { return height; }


    void print() const {
        for (int y = 0; y < height; ++y) {
            for (int x = 0; x < width; ++x) {
                std::cout << entries[y][x] << " ";
            }
            std::cout << "\n";
        }
    }
};

#endif
```

Annex B. Component.h

```
#ifndef COMPONENT_H
#define COMPONENT_H
#include <string>
#include <memory>
#include "Grid.h"

using namespace std;
struct Pad{
   int Px,Py;
};

class Component : public Grid<string> {
protected:
    int id;
    int x, y;
    string symbol;
    bool isOverlapping = false;

    static const int Maxpad = 20;
    Pad pads[Maxpad];
    int padcount;
    int getId() const;

public:
    Component(const int& id_, const int& x_, const int& y_,
              const int& width_, const int& height_,
              const string& sym_);

    virtual ~Component() {}
    virtual void drawonGrid(Grid<string>& comp);
    virtual std::unique_ptr<Component> clone() const = 0;
    string getName() const;
    int getX() const;
    int getY() const;

    void setOverlap(bool v) { isOverlapping = v; }
    bool getOverlap() const { return isOverlapping; }
    void setPosition(double nx, double ny);
    bool overlaps(const Component& other) const;
    bool violatesClearance(const Component& other, int
         minClearance) const;
    double distanceTo(const Component& other) const;
    bool contains(double px, double py) const;

    void addpad(int rx_, int ry_);
    bool getPadPosition(int index, int& outX, int& outY)
         const;
    int PadCount() const;
};
class Resistor : public Component {
public:
    Resistor(int id_, int x, int y);
    void drawonGrid(Grid<string>& comp) override;
    std::unique_ptr<Component> clone() const override;
};

class Capacitor : public Component {
public:
    Capacitor(int id_, int x, int y);
    void drawonGrid(Grid<string>& comp) override;
    std::unique_ptr<Component> clone() const override;
};

class Diode : public Component {
public:
    Diode(int id_, int x, int y);
    void drawonGrid(Grid<string>& comp) override;
    std::unique_ptr<Component> clone() const override;
};

class OpAmp : public Component {
public:
    OpAmp(int id_, int x, int y);
    void drawonGrid(Grid<string>& comp) override;
    std::unique_ptr<Component> clone() const override;
};

class Pin : public Component {
public:
    Pin(int id_, int x, int y, int numPins);
    void drawonGrid(Grid<string>& comp) override;
    std::unique_ptr<Component> clone() const override;
};

class Transistor : public Component {
public:
    Transistor(int id_, int x, int y);
    void drawonGrid(Grid<string>& comp) override;
    std::unique_ptr<Component> clone() const override;
};

#endif
```

Annex C. Component.cpp

```
#include "Component.h"
#include <iostream>
#include <cmath>
#include <algorithm>

using namespace std;
Component::Component(const int& id_, const int& x_, const
     int& y_,
                     const int& width_, const int& height_,
                          const string& sym_)
    : Grid<string>(width_, height_), id(id_), x(x_),
         y(y_), symbol(sym_)
{
    this->padcount = 0;
    for(int i = 0; i < getHeight(); i++) {
        for(int j = 0; j < getWidth(); j++) {
            (*this)(j, i) = symbol;
        }
    }
    if(getWidth() > 0 && getHeight() > 0) {
        (*this)(getWidth()/2, getHeight()/2) =
             to_string(id);
    }
}

void Component::addpad(int rx_, int ry_){
    if (padcount<Maxpad){
        pads[padcount].Px=rx_;
        pads[padcount].Py=ry_;
        padcount++;
    }else{
        cout<<"Error:"<<id<<"has too many pads"<<endl;
    }
}

//function to calculate the absolute position of the pad
     on the component
bool Component::getPadPosition(int index, int& outX, int&
     outY) const{
  if (index >=0 && index<padcount){
    outX = this->x + pads[index].Px;
    outY = this->y + pads[index].Py;
    return true;
```

```
  }
   return false;
}
int Component::PadCount() const{
    return padcount;
}

string Component::getName() const { return symbol; }
int Component::getX() const { return x; }
int Component::getY() const { return y; }

void Component::setPosition(double nx, double ny) {
    x = (int)nx; y = (int)ny;
}
int Component::getId() const {return id;}

void Component::drawonGrid(Grid<string>& comp) {
   int w = getWidth();
   int h = getHeight();
   int startX = x - (w / 2);
   int startY = y - (h / 2);

   for (int i = 0; i < h; ++i) {
      for (int j = 0; j < w; ++j) {
         int gridX = startX + j;
         int gridY = startY + i;
         if (gridX >= 0 && gridX < comp.getWidth() &&
             gridY >= 0 && gridY < comp.getHeight()) {
              comp(gridX, gridY) = (*this)(j, i);
         }
      }
   }
   //add pads
   for(int i=0; i < padcount; i++) {
        int px = x + pads[i].Px;
        int py = y + pads[i].Py;
        if (px >=0 && py >=0 && px < comp.getWidth() && py
            < comp.getHeight()) {
           comp(px, py) = "o";
        }
    }
}

bool Component::overlaps(const Component& other) const {
    int w1 = getWidth(), h1 = getHeight();
    int w2 = other.getWidth(), h2 = other.getHeight();
    int l1 = x - w1/2, r1 = l1 + w1;
    int t1 = y - h1/2, b1 = t1 + h1;
    int l2 = other.x - w2/2, r2 = l2 + w2;
    int t2 = other.y - h2/2, b2 = t2 + h2;
    return !(l1 >= r2 || l2 >= r1 || t1 >= b2 || t2 >= b1);
}

bool Component::violatesClearance(const Component& other,
     int minClearance) const {
    int w1 = getWidth(), h1 = getHeight();
    int w2 = other.getWidth(), h2 = other.getHeight();
    int l1 = x - w1/2 - minClearance, r1 = x + w1/2 +
        minClearance;
    int t1 = y - h1/2 - minClearance, b1 = y + h1/2 +
        minClearance;
    int l2 = other.x - w2/2 - minClearance, r2 = other.x +
        w2/2 + minClearance;
    int t2 = other.y - h2/2 - minClearance, b2 = other.y +
        h2/2 + minClearance;
    return !(l1 >= r2 || l2 >= r1 || t1 >= b2 || t2 >= b1);
}

double Component::distanceTo(const Component& other) const
    {
    return std::sqrt(std::pow(x - other.x, 2) + std::pow(y
        - other.y, 2));
}

bool Component::contains(double px, double py) const {
    int w = getWidth(), h = getHeight();
    int left = x - w/2;
    int top = y - h/2;
    return (px >= left && px < left + w && py >= top && py
        < top + h);
}

Resistor::Resistor(int id_, int x, int y) : Component(id_,
     x, y, 5, 1, "R") {
    addpad(-2, 0);
    addpad(2, 0);
}
void Resistor::drawonGrid(Grid<string>& comp) {
     Component::drawonGrid(comp); }
unique_ptr<Component> Resistor::clone() const { return
     make_unique<Resistor>(*this); }

Capacitor::Capacitor(int id_, int x, int y) :
     Component(id_, x, y, 5, 5, "C") {
    addpad(-2, 0);
    addpad(2, 0);
}
void Capacitor::drawonGrid(Grid<string>& comp) {
    int w = getWidth();
    int h = getHeight();
    double radiusSq = pow(min(w, h) / 2.0, 2);
    int startX = x - (w / 2);
    int startY = y - (h / 2);

    for (int i = 0; i < h; i++) {
        for (int j = 0; j < w; j++) {
            int gridX = startX + j;
            int gridY = startY + i;
            if (gridX >= 0 && gridX < comp.getWidth() &&
                 gridY >= 0 && gridY < comp.getHeight()) {
                double distSq = pow(gridX - x, 2) +
                     pow(gridY - y, 2);
                if (distSq <= radiusSq) {
                    if (gridX == x && gridY == y)
                         comp(gridX, gridY) = to_string(id);
                    else comp(gridX, gridY) = symbol;
                }
            }
        }
    }
}
unique_ptr<Component> Capacitor::clone() const { return
     make_unique<Capacitor>(*this); }

Diode::Diode(int id_, int x, int y) : Component(id_, x, y,
     5, 1, "D") {
    addpad(-2, 0);
    addpad(3, 0);
}
void Diode::drawonGrid(Grid<string>& grid) {
    int w = getWidth();
    int h = getHeight();
    int startX = x - (w / 2);
    int startY = y - (h / 2);
    for (int i = 0; i < h; ++i) {
        for (int j = 0; j < w; ++j) {
            int gridX = startX + j;
            int gridY = startY + i;
            if (gridX >= 0 && gridX < grid.getWidth() &&
                 gridY >= 0 && gridY < grid.getHeight()) {
                if (j == w - 1) grid(gridX, gridY) = "|";
```

```
                else if (gridX == x && gridY == y)
                    grid(gridX, gridY) = to_string(id);
                else grid(gridX, gridY) = "D";
            }
        }
    }
    //draw pads
    for(int i = 0; i < padcount; i++) {
        int px = x + pads[i].Px;
        int py = y + pads[i].Py;
        if (px >= 0 && py >= 0 && px < grid.getWidth() &&
            py < grid.getHeight()) {
            grid(px, py) = "o";
        }
    }
}
unique_ptr<Component> Diode::clone() const { return
    make_unique<Diode>(*this); }

OpAmp::OpAmp(int id_, int x_, int y_) : Component(id_, x_,
    y_, 4, 6, "O") {
    addpad(-3, -2);
    addpad(-3, -1);
    addpad(-3, 0);
    addpad(-3, 1);

    addpad( 2, 1);
    addpad( 2, 0);
    addpad( 2, -1);
    addpad( 2, -2);
}
void OpAmp::drawonGrid(Grid<string>& comp) {
    int w = getWidth();
    int h = getHeight();
    int startX = x - (w / 2);
    int startY = y - (h / 2);
    for (int i = 0; i < h; ++i) {
        for (int j = 0; j < w; ++j) {
            int gridX = startX + j;
            int gridY = startY + i;
            if (gridX >= 0 && gridX < comp.getWidth() &&
                gridY >= 0 && gridY < comp.getHeight()) {
                if (gridX == x && gridY == y) comp(gridX,
                    gridY) = to_string(id);
                else if (i == 0 && j == (w/2)) comp(gridX,
                    gridY) = "^";
                else comp(gridX, gridY) = symbol;
            }
        }
    }
    for(int i = 0; i < padcount; i++) {
        int px = x + pads[i].Px;
        int py = y + pads[i].Py;

        if (px >= 0 && py >= 0 && px < comp.getWidth() &&
            py < comp.getHeight()) {
            comp(px, py) = "o";
        }
    }
}
unique_ptr<Component> OpAmp::clone() const { return
    make_unique<OpAmp>(*this); }

Pin::Pin(int id_, int x_, int y_, int numPins) :
    Component(id_, x_, y_, 1, numPins, "P") {
    for (int i = 0; i < numPins; i++) {
        int relativeY = i - (numPins / 2);
        addpad(0, relativeY);
    }
}
void Pin::drawonGrid(Grid<string>& comp) {
    Component::drawonGrid(comp); }
unique_ptr<Component> Pin::clone() const { return
    make_unique<Pin>(*this); }

Transistor::Transistor(int id_, int x, int y) :
    Component(id_, x, y, 3, 3, "T") {
    addpad(-1, 0);
    addpad(0, 0);
    addpad(1, 0);
}
void Transistor::drawonGrid(Grid<string>& comp) {
    Component::drawonGrid(comp); }
unique_ptr<Component> Transistor::clone() const { return
    make_unique<Transistor>(*this); }
```

Annex D. Trace.h

```
#ifndef TRACE_H
#define TRACE_H
#include <string>
#include "Grid.h"

struct Point {
    int x, y;
    Point(int _x = 0, int _y = 0) : x(_x), y(_y) {}
};

class Trace {
protected:
   int netID;
   int layer;
   bool selected;

public:
   Trace(const int& netID_, const int& lay);
   virtual ~Trace();

   int getLayer() const;
   int getnetID() const;
   void setselec(bool s);

   virtual void drawonGrid(Grid<std::string>& grid) const
       = 0;
   virtual bool contains(Point point) const = 0;
};

#endif
```

Annex E. Trace.cpp

```
#include "Trace.h"

Trace::Trace(const int& netID_, const int& lay)
    : netID(netID_), layer(lay), selected(false)
{}
Trace::~Trace() {}
int Trace::getLayer() const { return layer; }
int Trace::getnetID() const { return netID; }
void Trace::setselec(bool s) { selected = s; }
```

Annex F. Lineartrace.h

```
    #ifndef LINEARTRACE_H
#define LINEARTRACE_H
#include "Trace.h"

class Lineartrace : public Trace {
private:
   Point* points;
   int pointCount;
   float thickness;

public:

   Lineartrace(int net_, int lay_, float thick, Point*
        pts, int count);
   ~Lineartrace();
   Point* getPoints() const { return points; }
   int getPointCount() const { return pointCount; }
   void drawonGrid(Grid<std::string>& grid) const override;
   bool contains(Point point) const override;
};

#endif
```

Annex G. Lineartrace.cpp

```
#include "Lineartrace.h"
#include <cmath>
#include <cstdlib>
using namespace std;

Lineartrace::Lineartrace(int net_, int lay_, float thick,
     Point* pts, int count)
    : Trace(net_, lay_), thickness(thick),
         pointCount(count)
{
    points = new Point[count];
    for(int i=0; i<count; i++) points[i] = pts[i];
}

Lineartrace::~Lineartrace() {
    delete[] points;
}

void Lineartrace::drawonGrid(Grid<std::string>& grid)
     const {
    if (pointCount < 2) return;

    for (int i = 0; i < pointCount - 1; ++i) {
        int x0 = points[i].x; int y0 = points[i].y;
        int x1 = points[i+1].x; int y1 = points[i+1].y;

        int dx = abs(x1 - x0);
        int sx = x0 < x1 ? 1 : -1;
        int dy = -abs(y1 - y0);
        int sy = y0 < y1 ? 1 : -1;
        int err = dx + dy;

        while (true) {
            if (x0 >= 0 && x0 < grid.getWidth() && y0 >= 0
                 && y0 < grid.getHeight()) {
                if (grid(x0, y0) == ".") grid(x0, y0) = "+";
            }
            if (x0 == x1 && y0 == y1) break;
            int e2 = 2 * err;
            if (e2 >= dy) { err += dy; x0 += sx; }
            if (e2 <= dx) { err += dx; y0 += sy; }
        }
    }
}

double pointToSegmentDist(Point p, Point a, Point b) {
    double l2 = pow(a.x - b.x, 2) + pow(a.y - b.y, 2);
    if (l2 == 0) return sqrt(pow(p.x - a.x, 2) + pow(p.y -
         a.y, 2));

    double t = ((p.x - a.x) * (b.x - a.x) + (p.y - a.y) *
         (b.y - a.y)) / l2;
    if (t < 0) t = 0;
    if (t > 1) t = 1;

    double projX = a.x + t * (b.x - a.x);
    double projY = a.y + t * (b.y - a.y);

    return sqrt(pow(p.x - projX, 2) + pow(p.y - projY, 2));
}

bool Lineartrace::contains(Point point) const {
    for (int i = 0; i < pointCount - 1; i++) {
        double dist = pointToSegmentDist(point, points[i],
             points[i+1]);
        if (dist < 0.8) {
            return true;
        }
    }
    return false;
}
```

Annex H. Validator.h

```
#ifndef VALIDATOR_H
#define VALIDATOR_H
#include <string>
class PCB;

class Validation {
protected:
    bool checker;
    std::string errormessage;

public:
    Validation();
    virtual ~Validation();
    virtual bool validate() = 0;

    bool getHasError() const;
    std::string getErrorMess() const;
    void reset();
};

class Overlap : public Validation {
private:
    PCB* board;
    int minClearance;
    bool checkOverlaps();
    bool checkClearances();
    bool checkBoundaries();

public:
    Overlap(PCB* pcb, int clearance = 1);
    bool validate() override;
};

#endif
```

Annex I. Validator.cpp

```
#include "Validator.h"
#include "PCB.h"
#include "Component.h"
#include <iostream>

using namespace std;
Validation::Validation() : checker(false),
     errormessage("") {}
Validation::~Validation() {}
bool Validation::getHasError() const {
    return checker;
}
std::string Validation::getErrorMess() const {
    return errormessage;
}

void Validation::reset() {
    checker = false;
    errormessage = "";
}
Overlap::Overlap(PCB* pcb, int clearance)
    : board(pcb), minClearance(clearance)
{
    checker = false;
    errormessage = "";
}

bool Overlap::validate() {
    reset();
    bool error = false;

    if (checkOverlaps()) error = true;
    if (checkBoundaries()) error = true;
    if (minClearance > 0 && checkClearances()) error =
         true;

    if (error) {
        checker = true;
        return false;
    }
    return true;
}

bool Overlap::checkOverlaps() {
    Component** comps = board->getComponents();
    int count = board->getComponentCount();
    for (int i = 0; i < count; i++) {
        comps[i]->setOverlap(false);
    }

    bool errorFound = false;

    for (int i = 0; i < count; i++) {
        for (int j = i + 1; j < count; j++) {

            if (comps[i]->overlaps(*comps[j])) {
                comps[i]->setOverlap(true);
                comps[j]->setOverlap(true);
                errormessage = "Overlap detected between " +
                               comps[i]->getName() + "(ID:" +
                                    to_string(comps[i]->getId())
                                    +")"
                               + " and " +
                               comps[j]->getName() + "(ID:" +
                                    to_string(comps[j]->getId())
                                    +")";
                cout << "ERROR: " << errormessage << endl;
                errorFound = true;
            }
        }
    }
    return errorFound;
}
bool Overlap::checkBoundaries() {
    Component** comps = board->getComponents();
    int count = board->getComponentCount();
    int boardW = board->getWidth();
    int boardH = board->getHeight();

    bool errorFound = false;

    for (int i = 0; i < count; i++) {
        int w = comps[i]->getWidth();
        int h = comps[i]->getHeight();

        int left = comps[i]->getX() - w/2;
        int right = left + w;
        int top = comps[i]->getY() - h/2;
        int bottom = top + h;

        if (left < 0 || right > boardW || top < 0 || bottom
             > boardH) {
            cout << "ERROR: Component " <<
                 comps[i]->getName()
                 << " (ID:" << comps[i]->getId() << ") is
                      outside the board!" << endl;
            errorFound = true;
        }
    }
    return errorFound;
}

bool Overlap::checkClearances() {
    Component** comps = board->getComponents();
    int count = board->getComponentCount();

    bool errorFound = false;

    for (int i = 0; i < count; i++) {
        for (int j = i + 1; j < count; j++) {

            if (comps[i]->violatesClearance(*comps[j],
                 minClearance)) {
                errormessage = "Clearance violation between
                     " +
                               comps[i]->getName() + "(ID:" +
                                    to_string(comps[i]->getId())
                                    +")"
                               + " and " +
                               comps[j]->getName() + "(ID:" +
                                    to_string(comps[j]->getId())
                                    +")";
                cout << "ERROR: " << errormessage << endl;
                errorFound = true;
            }
        }
    }
    return errorFound;
}
```

Annex J. PCB.h

```
#ifndef PCB_H
#define PCB_H
#include "Component.h"
#include "Trace.h"
#include "Validator.h"
#include <string>
```

```
class PCB {
private:

    Component** components;
    int compCount;
    int compCapacity;
    Trace** traces;
    int traceCount;
    int traceCapacity;
    Validation** validators;
    int valCount;
    int valCapacity;
    double boardWidth;
    double boardHeight;
    void resizeComponents();
    void resizeTraces();
    void resizeValidators();

public:
    PCB(const double& width, const double& height);
    ~PCB();

    void addComponent(Component* c);
    void addTrace(Trace* t);
    void addValidator(Validation* v);
    bool validateBoard();
    void printBoard();
    void setDimensions(double w, double h);

    Component** getComponents() const;
    int getComponentCount() const;
    Trace** getTraces() const;
    int getTraceCount() const;
    double getWidth() const;
    double getHeight() const;

    void removeComponent(int index);
    void removeTrace(int index);
    Component* getComponentAt(int x, int y) const;
    Trace* getTraceAt(int x, int y) const;
};

#endif
```

Annex K. PCB.cpp

```
#include "PCB.h"
#include "Validator.h"
#include <iostream>
#include <cassert>

using namespace std;

PCB::PCB(const double& width, const double& height)
    : boardWidth(width), boardHeight(height),
      compCount(0), compCapacity(2),
      traceCount(0), traceCapacity(2),
      valCount(0), valCapacity(2) {
    components = new Component*[compCapacity];
    traces = new Trace*[traceCapacity];
    validators = new Validation*[valCapacity];
    addValidator(new Overlap(this));
}

PCB::~PCB() {
    for (int i = 0; i < compCount; ++i) {
        delete components[i];
    }
    delete [] components;
    for (int i = 0; i < traceCount; ++i) {
        delete traces[i];
    }
    delete [] traces;
    for (int i = 0; i < valCount; ++i) {
        delete validators[i];
    }
    delete [] validators;
}
void PCB::resizeComponents() {
    compCapacity *= 2;
    Component** newArr = new Component*[compCapacity];
    for(int i=0; i<compCount; i++) newArr[i] =
         components[i];
    delete[] components;
    components = newArr;
}
void PCB::resizeTraces() {
    traceCapacity *= 2;
    Trace** newArr = new Trace*[traceCapacity];
    for(int i=0; i<traceCount; i++) newArr[i] = traces[i];
    delete[] traces;
    traces = newArr;
}
void PCB::resizeValidators() {
    valCapacity *= 2;
    Validation** newArr = new Validation*[valCapacity];
    for(int i=0; i<valCount; i++) newArr[i] =
         validators[i];
    delete[] validators;
    validators = newArr;
}
void PCB::addComponent(Component* c) {
    if (compCount == compCapacity) resizeComponents();
    components[compCount++] = c;
}

void PCB::addTrace(Trace* t) {
    if (traceCount == traceCapacity) resizeTraces();
    traces[traceCount++] = t;
}

void PCB::addValidator(Validation* v) {
    if (valCount == valCapacity) resizeValidators();
    validators[valCount++] = v;
}

bool PCB::validateBoard() {
    bool allPassed = true;
    cout << "Starting PCB Validation Check" <<endl;
    for (int i = 0; i < valCount; ++i) {
        if (!validators[i]->validate()) {
            allPassed = false;
            cout << "Violation: " <<
                 validators[i]->getErrorMess() << std::endl;
        }
    }
    if(allPassed) cout << "NO ERRORS FOUND" <<endl;
    else cout << "ERRORS DETECTED" <<endl;
    return allPassed;
}
void PCB::setDimensions(double w, double h){
    boardHeight=h;
    boardWidth=w;

}
void PCB::printBoard() {
    int gridW = (int)boardWidth;
    int gridH = (int)boardHeight;
    Grid<std::string> displayGrid(gridW, gridH);
```

```
    for(int y=0; y<gridH; y++)
        for(int x=0; x<gridW; x++) displayGrid(x, y) = ".";

    for (int i = 0; i < traceCount; ++i)
        if(traces[i]) traces[i]->drawonGrid(displayGrid);

    for (int i = 0; i < compCount; ++i)
        if(components[i])
             components[i]->drawonGrid(displayGrid);

    cout << "  ";
    for(int i=0; i<gridW; i++) cout << i%10;
    cout << "\n";
    for (int y=0; y<gridH; ++y) {
        cout << y%10 << " |";
        for (int x=0; x<gridW; ++x) cout << displayGrid(x,
             y);
        cout << "|\n";
    }
}
Component** PCB::getComponents() const{
    return components;
}
int PCB::getComponentCount() const{
    return compCount;
}
Trace** PCB::getTraces() const{
   return traces;
}
int PCB::getTraceCount() const{
   return traceCount;
}
double PCB::getWidth() const { return boardWidth; }
double PCB::getHeight() const { return boardHeight; }

void PCB::removeComponent(int index){
  assert(index>=0 && index<=compCount);
  delete components[index];
  for(int i=index;i<compCount-1;i++){
    components[i]=components[i+1];
  }
  compCount--;
}

void PCB::removeTrace(int index){
  assert(index>=0 && index<=traceCount);
  delete traces[index];
  for(int i=index;i<traceCount-1;i++){
    traces[i]=traces[i+1];
  }
  traceCount--;
}

Component* PCB::getComponentAt(int x, int y) const {
    for (int i = 0; i < compCount; i++) {
        if (components[i]->contains(x, y)) {
            return components[i];
        }
    }
    return nullptr;
}

Trace* PCB::getTraceAt(int x, int y) const {
    for (int i = 0; i < traceCount; i++) {
        if (traces[i]->contains(Point(x, y))) {
            return traces[i];
        }
    }
    return nullptr;
}
```

Annex L. PCBViewer.h

```
#ifndef PCBVIEWER_H
#define PCBVIEWER_H

#include <SFML/Graphics.hpp>
#include "PCB.h"
#include <vector>
#include <string>


class PCBViewer {
private:
    int scale;
    sf::Font font;

    sf::View boardView;

    const sf::Color COL_BG = sf::Color(20, 20, 20);
    const sf::Color COL_GRID = sf::Color(40, 40, 40);
    const sf::Color COL_TOP_COPPER = sf::Color(200, 0, 0);
    const sf::Color COL_PAD_CENTER = sf::Color(0, 150, 0);
    const sf::Color COL_PAD_RING = sf::Color(0, 200, 0);
    const sf::Color COL_SILKSCREEN = sf::Color(220, 220,
         220);

    void drawPad(sf::RenderWindow& window, float x, float
         y, bool isSquare = false);
    void drawResistorCAD(sf::RenderWindow& window, float
         x, float y, float w, float h, sf::Color color);
    void drawCapacitorCAD(sf::RenderWindow& window, float
         x, float y, float w, float h, sf::Color color);
    void drawDiodeCAD(sf::RenderWindow& window, float x,
         float y, float w, float h, sf::Color color);
    void drawTransistorCAD(sf::RenderWindow& window, float
         x, float y, float w, float h, sf::Color color);
    void drawOpAmpCAD(sf::RenderWindow& window, float x,
         float y, float w, float h, sf::Color color);
    void drawPinCAD(sf::RenderWindow& window, float x,
         float y, float w, float h, int numPins, sf::Color
         color);
    void drawTraceCAD(sf::RenderWindow& window, Trace* t);

    void fitToWindow(const sf::RenderWindow& window, const
         PCB& board);

public:
    PCBViewer(int scaleFactor = 20);
    void visualize(PCB& board);
    void drawBoard(sf::RenderWindow& window, PCB& board);
};

#endif
```

Annex M. PCBViewer.cpp

```
#include "PCBViewer.h"
#include <iostream>
#include <cmath>
#include "Lineartrace.h"
#include "Component.h"

PCBViewer::PCBViewer(int scaleFactor) : scale(scaleFactor)
     {
```

```
    if (!font.loadFromFile("arial.ttf")) {
        std::cout << "Warning: Font not found (arial.ttf)"
             << std::endl;
    }
}

void PCBViewer::drawPad(sf::RenderWindow& window, float x,
     float y, bool isSquare) {
    float radius = scale * 0.4f;
    sf::Shape* shape;
    sf::CircleShape circ(radius);
    sf::RectangleShape rect(sf::Vector2f(radius*2,
         radius*2));

    if (isSquare) {
        rect.setOrigin(radius, radius);
        rect.setPosition(x, y);
        shape = &rect;
    } else {
        circ.setOrigin(radius, radius);
        circ.setPosition(x, y);
        shape = &circ;
    }
    shape->setFillColor(COL_PAD_CENTER);
    shape->setOutlineThickness(2);
    shape->setOutlineColor(COL_PAD_RING);
    window.draw(*shape);

    sf::CircleShape hole(scale * 0.15f);
    hole.setOrigin(hole.getRadius(), hole.getRadius());
    hole.setPosition(x, y);
    hole.setFillColor(COL_BG);
    window.draw(hole);
}

void PCBViewer::drawResistorCAD(sf::RenderWindow& window,
     float x, float y, float w, float h, sf::Color color) {
    drawPad(window, x - w/2 + scale/2.0f, y);
    drawPad(window, x + w/2 - scale/2.0f, y);
    sf::RectangleShape body(sf::Vector2f(w - scale*1.2f, h
         * 0.6f));
    body.setOrigin(body.getSize().x/2, body.getSize().y/2);
    body.setPosition(x, y);
    body.setFillColor(sf::Color::Transparent);
    body.setOutlineColor(color);
    body.setOutlineThickness(2);
    window.draw(body);
}

void PCBViewer::drawCapacitorCAD(sf::RenderWindow& window,
     float x, float y, float w, float h, sf::Color color) {
    drawPad(window, x - scale, y);
    drawPad(window, x + scale, y);
    sf::CircleShape body(scale * 0.8f);
    body.setOrigin(body.getRadius(), body.getRadius());
    body.setPosition(x, y);
    body.setFillColor(sf::Color::Transparent);
    body.setOutlineColor(color);
    body.setOutlineThickness(2);
    window.draw(body);
}

void PCBViewer::drawDiodeCAD(sf::RenderWindow& window,
     float x, float y, float w, float h, sf::Color color) {

    float centerX = x + scale;
    drawPad(window, x- w/2 + scale/2.0f, y, true);
    drawPad(window, centerX + w/2 - scale/2.0f, y);

    float bw = w - scale * 1.2f;
    sf::RectangleShape body(sf::Vector2f(bw, h * 0.6f));
    body.setOrigin(bw / 2, h * 0.3f);
    body.setPosition(centerX-scale/2, y);
    body.setFillColor(sf::Color::Transparent);
    body.setOutlineColor(color);
    body.setOutlineThickness(2);
    window.draw(body);
    sf::RectangleShape line(sf::Vector2f(2, h * 0.6f));
    line.setOrigin(1, h * 0.3f);
    line.setPosition(centerX + bw * 0.3f, y);
    line.setFillColor(color);
    window.draw(line);
}

void PCBViewer::drawTransistorCAD(sf::RenderWindow&
     window, float x, float y, float w, float h, sf::Color
     color) {
    drawPad(window, x - scale, y);
    drawPad(window, x, y);
    drawPad(window, x + scale, y);
    float r = scale * 0.9f;
    sf::CircleShape body(r);
    body.setOrigin(r, r);
    body.setPosition(x, y - scale*0.3f);
    body.setFillColor(sf::Color::Transparent);
    body.setOutlineColor(color);
    body.setOutlineThickness(2);
    window.draw(body);
}

void PCBViewer::drawOpAmpCAD(sf::RenderWindow& window,
     float x, float y, float w, float h, sf::Color color) {
    // Body
    sf::RectangleShape body(sf::Vector2f(3.0f *
         scale+scale, 4.0f * scale));
    body.setOrigin(1.5f * scale, 2.0f * scale);
    body.setPosition(x-scale, y - 0.5f * scale);
    body.setFillColor(sf::Color(50, 50, 50));
    body.setOutlineColor(color);
    body.setOutlineThickness(2);
    window.draw(body);

    // Pads
    for (int i = 0; i < 4; i++) {
        float py = y + (i - 2) * scale;
        drawPad(window, x-scale - 2 * scale, py, i == 3);
        drawPad(window, x + 2 * scale, py);
    }


    // Notch
    sf::CircleShape n(scale * 0.2f);
    n.setOrigin(n.getRadius(), n.getRadius());
    n.setPosition(x-scale, y - 2.5f * scale);
    n.setFillColor(sf::Color::White);
    window.draw(n);
}

void PCBViewer::drawPinCAD(sf::RenderWindow& window, float
     x, float y, float w, float h, int numPins,sf::Color
     color) {
    float startY;
    if (numPins % 2 != 0) startY = y - ((numPins - 1) / 2)
         * scale;
    else startY = y - ((numPins - 1) * scale) / 2.0f-(0.5f
         * scale);

    for(int i=0; i<numPins; i++) {
        drawPad(window, x, startY + (i*scale), (i==0));
    }

    float totalH = numPins * scale;
```

```
    sf::RectangleShape box(sf::Vector2f(scale, totalH));
    float visualCenterY = startY + ((numPins-1) * scale) /
         2.0f;
    box.setOrigin(scale/2, totalH/2);
    box.setPosition(x, visualCenterY);
    box.setFillColor(sf::Color::Transparent);
    box.setOutlineColor(COL_SILKSCREEN);
    box.setOutlineThickness(2);
    window.draw(box);
}

void PCBViewer::drawTraceCAD(sf::RenderWindow& window,
     Trace* t) {
    Lineartrace* linear = dynamic_cast<Lineartrace*>(t);
    if (!linear || linear->getPointCount() < 2) return;

    Point* pts = linear->getPoints();
    float thick = scale * 0.3f;
    if (thick < 1.0f) thick = 1.0f;
    sf::Color traceColor = COL_TOP_COPPER;

    for (int i = 0; i < linear->getPointCount() - 1; i++) {
        sf::Vector2f p1(pts[i].x * scale, pts[i].y * scale);
        sf::Vector2f p2(pts[i+1].x * scale, pts[i+1].y *
             scale);
        sf::Vector2f diff = p2 - p1;
        float len = std::sqrt(diff.x*diff.x +
             diff.y*diff.y);
        float angle = std::atan2(diff.y, diff.x) * 180.f /
             3.14159f;

        sf::RectangleShape line(sf::Vector2f(len, thick));
        line.setOrigin(0, thick/2);
        line.setPosition(p1);
        line.setRotation(angle);
        line.setFillColor(COL_TOP_COPPER);
        window.draw(line);

        sf::CircleShape joint(thick/2);
        joint.setOrigin(thick/2, thick/2);
        joint.setPosition(p1);
        joint.setFillColor(COL_TOP_COPPER);
        window.draw(joint);
    }

    sf::CircleShape endJoint(thick/2);
    endJoint.setOrigin(thick/2, thick/2);
    endJoint.setPosition(pts[linear->getPointCount()-1].x
         * scale, pts[linear->getPointCount()-1].y * scale);
    endJoint.setFillColor(COL_TOP_COPPER);
    window.draw(endJoint);
}

void PCBViewer::fitToWindow(const sf::RenderWindow&
     window, const PCB& board) {
    float boardW = board.getWidth() * scale;
    float boardH = board.getHeight() * scale;

    boardView.setCenter(boardW / 2.0f, boardH / 2.0f);
    boardView.setSize(window.getSize().x,
         window.getSize().y);

    float ratioX = boardW / window.getSize().x;
    float ratioY = boardH / window.getSize().y;
    float zoomFactor = (ratioX > ratioY) ? ratioX : ratioY;

    if (zoomFactor < 1.0f) zoomFactor = 1.0f;
    boardView.zoom(zoomFactor * 1.1f);
}

void PCBViewer::visualize(PCB& board) {
    sf::RenderWindow window(sf::VideoMode(1200, 800), "PCB
         Final Viewer");
    fitToWindow(window, board);

    while (window.isOpen()) {
        sf::Event event;
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed)
                window.close();
            if (event.type == sf::Event::Resized) {
                boardView.setSize(event.size.width,
                     event.size.height);
                fitToWindow(window, board);
            }
        }

        window.clear(sf::Color::Black);
        window.setView(boardView);
        drawBoard(window, board);

        window.display();
    }
}

void PCBViewer::drawBoard(sf::RenderWindow& window, PCB&
     board) {
    sf::RectangleShape bg(sf::Vector2f(board.getWidth() *
         scale, board.getHeight() * scale));
    bg.setFillColor(COL_BG);
    window.draw(bg);

    sf::Vertex line[2];
    for (int i = 0; i <= board.getWidth(); i++) {
        line[0] = sf::Vertex(sf::Vector2f(i * scale, 0),
             COL_GRID);
        line[1] = sf::Vertex(sf::Vector2f(i * scale,
             board.getHeight() * scale), COL_GRID);
        window.draw(line, 2, sf::Lines);
    }
    for (int i = 0; i <= board.getHeight(); i++) {
        line[0] = sf::Vertex(sf::Vector2f(0, i * scale),
             COL_GRID);
        line[1] = sf::Vertex(sf::Vector2f(board.getWidth()
             * scale, i * scale), COL_GRID);
        window.draw(line, 2, sf::Lines);
    }

    Trace** traces = board.getTraces();
    for (int i = 0; i < board.getTraceCount(); i++)
         drawTraceCAD(window, traces[i]);

    Component** comps = board.getComponents();
    for (int i = 0; i < board.getComponentCount(); i++) {
        Component* c = comps[i];
        float x = c->getX() * scale;
        float y = c->getY() * scale;
        float w = c->getWidth() * scale;
        float h = c->getHeight() * scale;

        sf::Color outlineColor = c->getOverlap() ?
             sf::Color::Red : COL_SILKSCREEN;

        if (dynamic_cast<Resistor*>(c))
            drawResistorCAD(window, x, y, w, h,
                 outlineColor);
        else if (dynamic_cast<Capacitor*>(c))
            drawCapacitorCAD(window, x, y, w, h,
                 outlineColor);
        else if (dynamic_cast<Diode*>(c))
            drawDiodeCAD(window, x, y, w, h, outlineColor);
```

```
        else if (dynamic_cast<Transistor*>(c))
            drawTransistorCAD(window, x, y, w, h,
                outlineColor);
        else if (dynamic_cast<OpAmp*>(c))
            drawOpAmpCAD(window, x, y, w, h, outlineColor);
        else if (dynamic_cast<Pin*>(c))
            drawPinCAD(window, x, y, w, h, c->getHeight(),
                outlineColor);
    }
}
```

Annex N. Main.cpp

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <map>
#include <vector>
#include "PCB.h"
#include "Validator.h"
#include "Trace.h"
#include "Lineartrace.h"
#include "PCBViewer.h"
#include "Component.h"

using namespace std;

void loadProject(PCB& myBoard, const string& compFile,
     const string& traceFile) {
    ifstream cFile(compFile);
    if (!cFile.is_open()) {
        cerr << "Error: Could not open " << compFile <<
            endl;
        return;
    }
    string line;
    if (getline(cFile, line)) {
        stringstream ss(line);
        string label;
        int w, h;
        if (ss >> label >> w >> h && label == "Dimension") {
            myBoard.setDimensions(w, h);
            cFile.clear();
            cFile.seekg(0);
        }
    }
    map<string, int> typeCounters;

    while (getline(cFile, line)) {
        if (line.empty()) continue;
        stringstream ss(line);
        string type;
        ss >> type;
        int autoID = ++typeCounters[type];

        int x, y, extra;
        if (type == "Resistor") {
            ss >> x >> y;
            myBoard.addComponent(new Resistor(autoID, x,
                y));
        } else if (type == "Pin") {
            ss >> x >> y >> extra;
            myBoard.addComponent(new Pin(autoID, x, y,
                extra));
        } else if (type == "Diode") {
            ss >> x >> y;
            myBoard.addComponent(new Diode(autoID, x, y));
        } else if (type == "Transistor") {
            ss >> x >> y;
            myBoard.addComponent(new Transistor(autoID, x,
                y));
        } else if (type == "OpAmp") {
            ss >> x >> y;
            myBoard.addComponent(new OpAmp(autoID, x, y));
        }
    }
    ifstream tFile(traceFile);
    int traceCounter = 1;

    while (getline(tFile, line)) {
        if (line.empty()) continue;
        stringstream ss(line);

        float thick;
        ss >> thick;
        vector<Point> pts;
        int px, py;
        while (ss >> px >> py) {
            pts.push_back(Point(px, py));
        }

        if (!pts.empty()) {
            myBoard.addTrace(new
                Lineartrace(traceCounter++, 1, thick,
                pts.data(), (int)pts.size()));
        }
    }
}
int main() {
    PCB myBoard(0, 0);
    cout << "--- Initializing Components ---" << endl;
    loadProject(myBoard, "Component.txt", "Traces.txt");

    cout << "--- Validating Board ---" << endl;
    myBoard.validateBoard();

    cout << "\n--- PCB Grid View ---" << endl;
    myBoard.printBoard();

    cout << "\nOpening Graphical Viewer..." << endl;
    PCBViewer viewer(35);
    viewer.visualize(myBoard);

    return 0;
}
```