Project Manual

Digital Logic Simulator Java Swing application built by Nicolas Conrad and Kyle Pickle for our ECS 160 final project.

Project Report

Introduction

The goal of this project is, as stated in the requirements doc, to develop a Digital Logic Simulator for educational purposes, designed to help students understand and experiment with digital logic circuits. We seek to prove our knowledge of Software Engineering by providing an application that is organized, intuitive, and easy for ourselves or future developers to add on to.

Literature Review/Background Study

Our project most closely resembles logic simulating programs like Logisim, which is fairly well-received as an application. (need some more background/history here). I believe what sets our program apart is that it is “lighter” and more digital-based, abstracting binary signals as numbers and BitSets rather than positive/negative voltage.
Educational tools like this one greatly benefit from responsive colors, buttons, etc. when interacting with the app. Thus it is important for us to use these to our advantage to help people clearly visualize and debug their projects quickly and easily, perhaps more so than adding other extraneous features like rotations, scaling, etc. depending on our intended audience.
Tools like this one can be very educational for the very reason it is very simplistic - it becomes an exercise for users to make complex circuits from a set of fundamental logic gates.

Methodology

We spent most of the first week developing our class diagram and getting the repository set up with Git, Maven, and our respective IDEs. We then continued to split up tasks specific to the goals for each week, as seen in the following section. These tasks / user stories are split up by GitHub branch, typically one per group member at a given time. We communicate on Discord and are already partners for our senior project this year, which is helpful.

Implementation Details

We do weekly checkups / version control on a weekly basis for the rest of the quarter, based on the weekly goals outlined in the requirements doc:

- **Week 1:** Planning and UI design, including the development of the logic gate palette and circuit workspace.

- **Week 2:** Implementation of logic gate placement, movement, and the wiring tool for connecting gates.
- **Week 3:** Development of the simulation mode to visualize signal flow and logic gate outputs, and implementation of save/load functionality.
- **Week 4:** Final testing, creation of user documentation, preparation of the demo video, and project wrap-up.

We are using Eclipse and VSCode and their respective plugins to make development easier. ChatGPT is also a vital tool to get this project completed in time, though it is getting progressively less helpful as our project grows.

Testing and Evaluation

Intermittently, we have been extensively debugging and identifying issues to each other to keep them in the know while we develop our application. Before we submit we plan on testing the application extensively, and if time allows will hopefully implement a test jar that automatically creates and tests sets of premade circuits.

Results and Discussion

TBD.

Conclusion

TBD.

References and Appendices

TBD. Will include sources used in the Literature/Background section.

User Manual

Upon opening the logic gate application, you will be presented with several different sections and panels to be used in the creation of your digital circuits.

The most prominent of all, is the gridded circuit workspace in the center. This is where you will place all of your logic gates and wires to connect them. This window is both draggable as well as zoomable, allowing for an infinite workspace size.

There are several options to help with movement within your workspace, such as zoom in, zoom out, select and drag. When “select” toggled, you will be able to press on different logic gates to select them, and configure its parameters in the logic gate parameters panel discussed later in the manual. When “drag” is toggled, you are able to drag your mouse to move through the circuit workspace, and move to sections of the circuit which may not have appeared in the window previously.

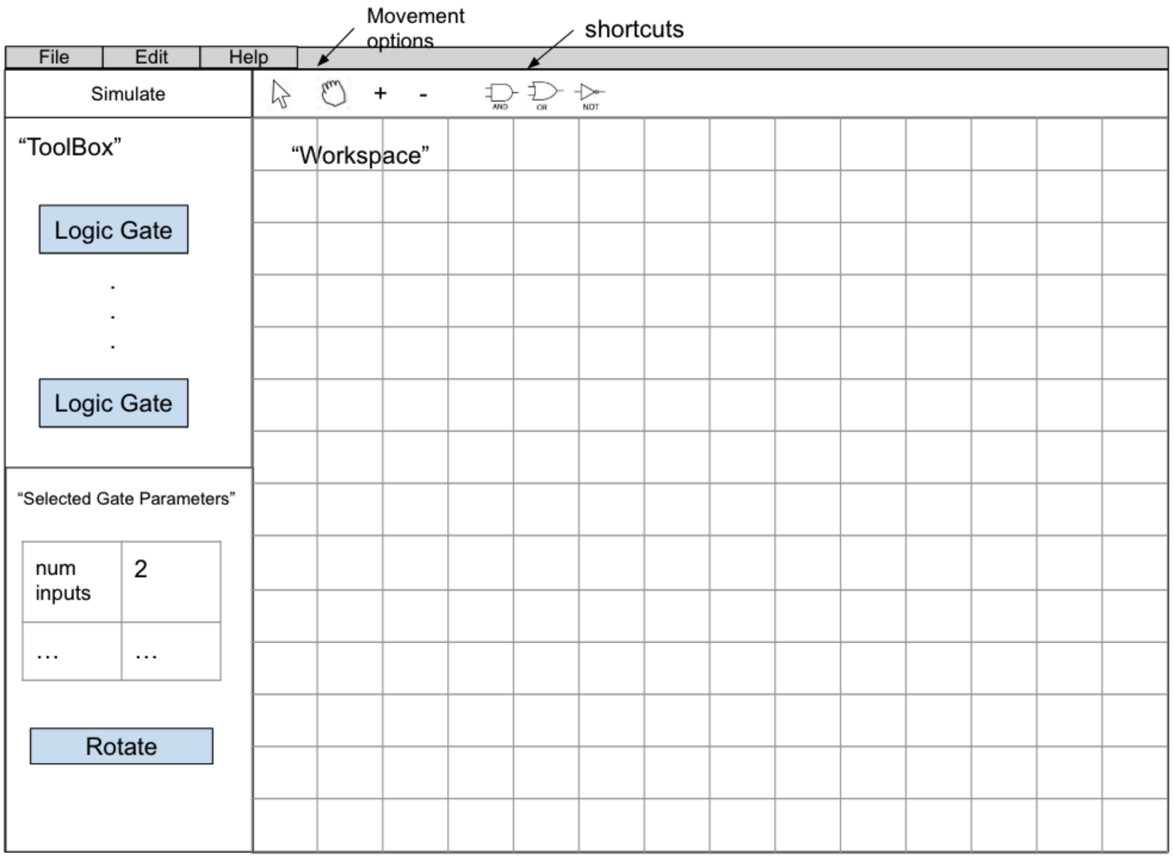
In order to find logic gates to place within the workspace, the upper panel on the western border of the application contains the logic gate palette. You may scroll through this panel to find the desired gate and simply drag it into the workspace to use it.

Below this, is a logic gate parameters panel. If a logic gate in the circuit workspace is selected, a table of parameters will pop up, allowing you to edit the configuration of a logic gate. Additionally, there is a rotate function, allowing you to rotate the logic gate as desired.

Once you have created a satisfactory circuit and wish to analyze its behavior, the simulate button can be used. This button is located in the same panel as the movement options. This will simulate the behavior of your created circuit. The signal flow and the output of each gate will be visually represented during the simulation process.

Lastly, on the top panel of the application are three menu options: File, Edit, and Help. File has two submenus: Save and Load. Save allows you to save the work you have completed in the circuit workspace, and load allows you to load a previously saved circuit into the workspace. Edit will provide several different options in regards to making changes to the workspace. Lastly, the Help menu will provide resources to guide you in the use of the application.

Initial UI Diagram

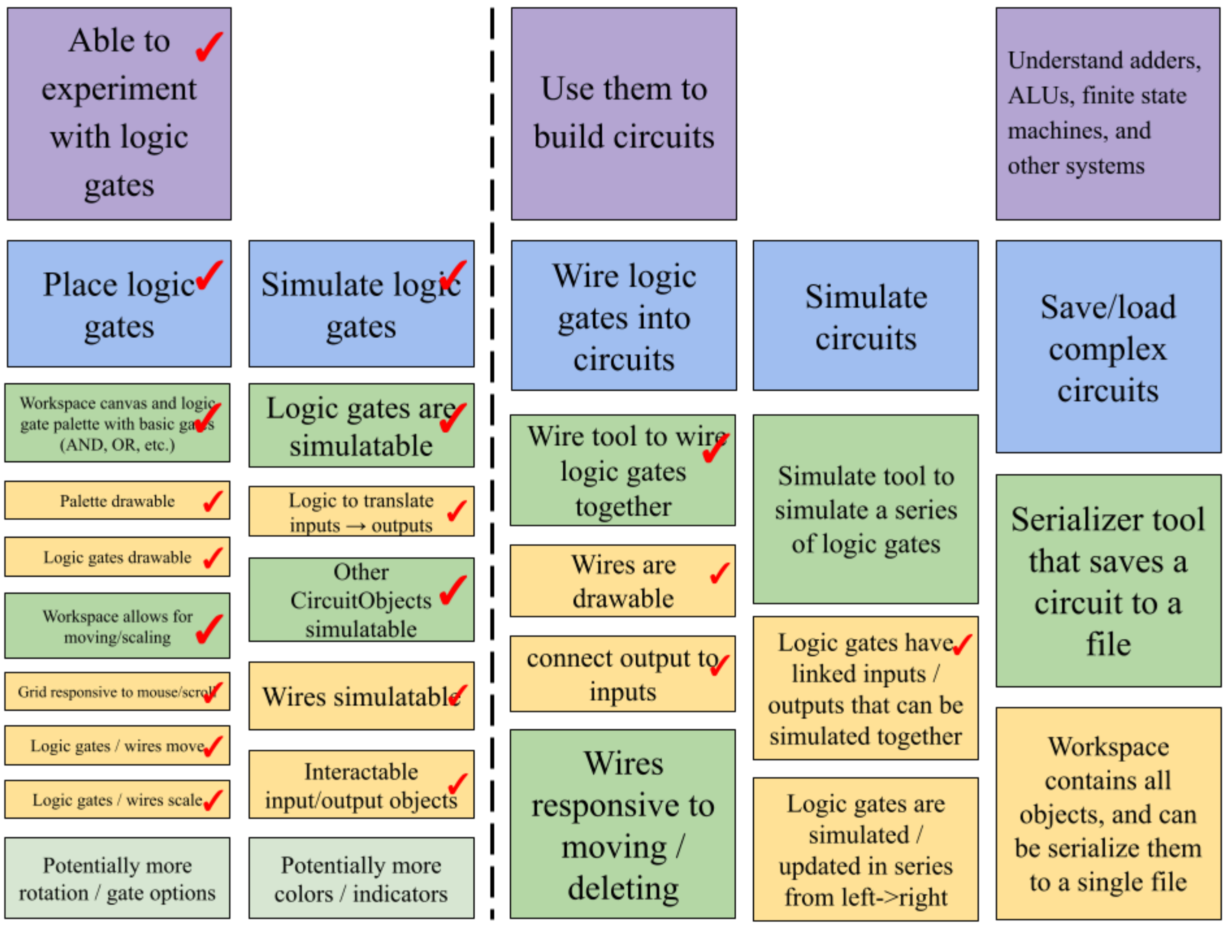


Software Design

BHAG

As prospecting computer architects, we want to be able to experiment with logic gates and use them to build circuits that will help us to understand adders, ALUs, finite state machines, and other systems that are essential parts of a computer.

User Stories



Architecture Overview

At a high level, we've chosen to model our project as closely to the project requirements as possible to make it very clear which parts are completed and which still need to be implemented, as well as to make it easy for ourselves, our TA, professor, etc. who know the requirements to easily find what they are looking for. As such, we plan on implementing the 5 "specific features" - the Logic Gate Palette, Circuit Workspace, Wiring Tool, Simulation Mode, and Circuit Save/Load Functionality - into 5 classes alongside App.java.

Underneath, the actual drawing, simulating, and serializing will be interfaced and split up as organized as possible to stay readable and allow for easy modification/addition. For a more detailed picture, refer to the class diagram below.

Design Patterns

Composite

There are multiple composite design patterns throughout our project. The App class is a composite that manages the 5 main singleton classes of our app. The Workspace class contains and manages a list of CircuitObjects built by the palette and WiringTool classes. Finally, each CircuitObject and class derived from it is composed of input and output nodes strung together through logic.

Factory

The Palette and WiringTool classes are builders of logic gates that generate new logic gates and wires for use in the Workspace, respectively. While in week 1 we had thought about using a LogicGateBuilder class, we believe that implementing small individual classes for each base logic gate is best, pushing as much functionality into the inherited CircuitObject class as possible.

Singleton

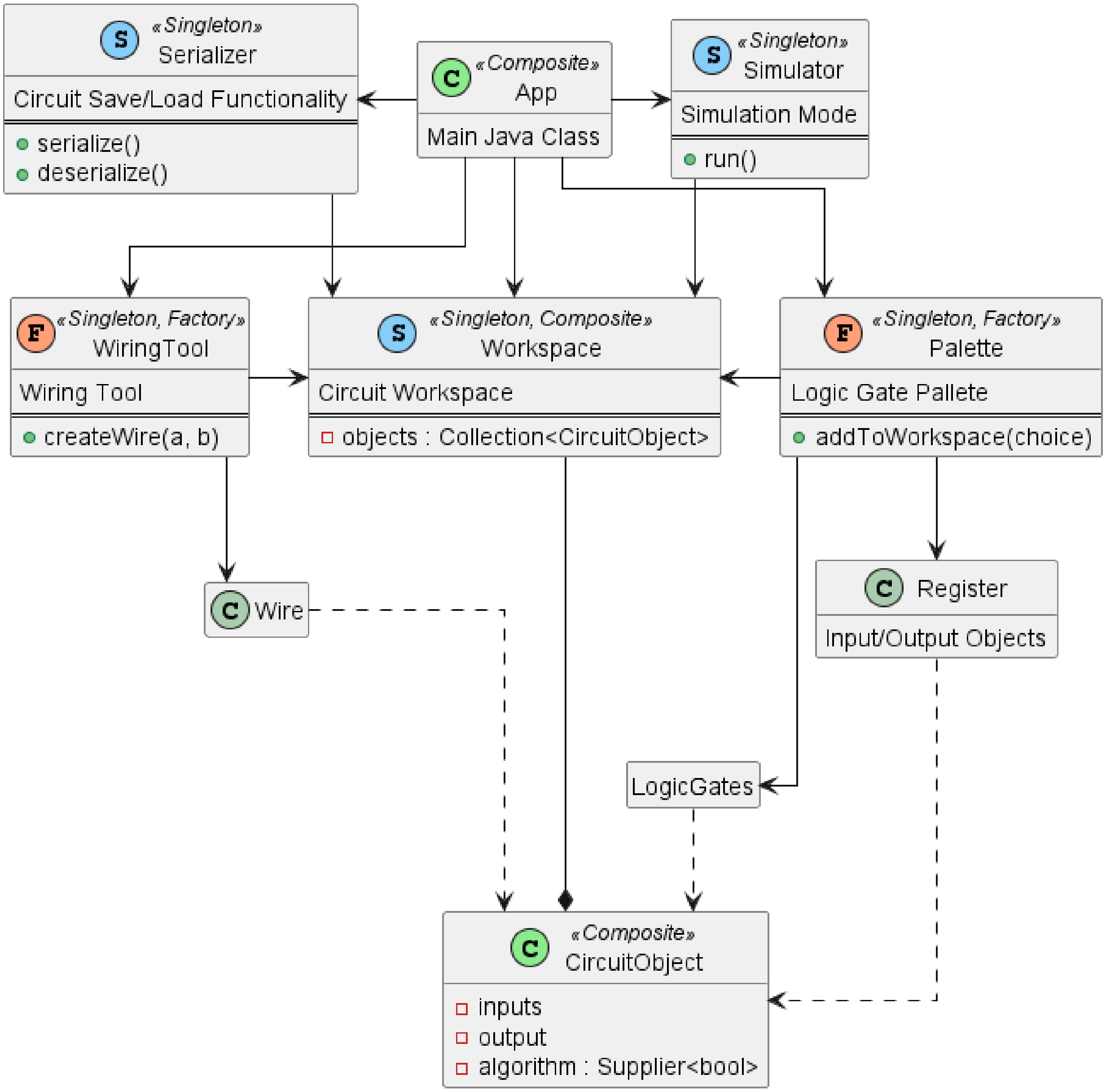
We plan on implementing singletons to represent the 5 mandatory "features" of our project. This will help with organization, readability, and make it easy to delegate functionality based on the project requirements.

Component Descriptions

Specific descriptions of each component are still TBD, but a rough overview of our plan can be found in the following Diagrams section.

Diagrams

The following is a rough class diagram agreed upon for our Digital Logic Simulator project, rendered with PlantUML (https://www.plantuml.com/plantuml/uml/). From Week 1, we decided to split up the LogicGateBuilder/LogicGates classes between the Palette and previously-mentioned individual logic gate (And, Or, etc.) classes. We also absorbed the interfaces we originally had planned and have instead implemented much of our functionality into the CircuitObject class to make it as easy as possible to make new objects down the line (which has greatly helped when creating the Register class and other inputs for debugging).



Standards and Conventions

We have been doing our best to follow the Google Java Styling Guide as a reference (https://google.github.io/styleguide/javaguide.html), though have been making it a priority to make the code between the both of us indistinguishable.

Appendix

Full ChatGPT HTMLs are located in our project bundle.

Chat GPT Logs for Nicolas Conrad

Week 1

Certainly! Here's a summary of our conversation:

1. **Objective:** You wanted to create a Digital Logic Simulator using Java Swing similar to Logisim, allowing users to create logic circuits with wires and gates.

2. **Class Structure (UML):** You provided a UML diagram with classes like `App`, `Palette`, `Workspace`, `WiringTool`, `Simulator`, `Serializer`, `Wire`, `Register`, `LogicGateBuilder`, `LogicGate`, `CircuitObject`, and interfaces `Serializable`, `Drawable`, `Simulatable`.

3. **Implementation:** I provided code snippets to implement the missing parts based on the UML structure. The `App` class acts as the main class with a GUI, and other classes handle logic gates, workspace, serialization, and simulation.

4. **Functionality Added:**
   - **Save/Load:** Added save and load submenus under the "File" menu using the `Serializer` class.
   - **Grid Background:** Added a grid background to the workspace for snapping wires and gates to the grid.
   - **Zoom and Drag:** Made the workspace draggable and zoomable using mouse events.

5. **Palette Adjustments:** Adjusted the palette layout to a 2-column grid for both logic gates and parameters.

6. **GUI Features:** Added simulate, zoom in, zoom out, and movement options in a toolbar. Also added a logic gate parameters panel with a rotate function.

7. **Issues:** Addressed compilation errors and implemented missing parts of the code.

8. **Recent Request:** You asked for the logic gate palette to be a 2-column grid.

Week 2

Certainly! Here's a summary of our conversation:

1. **Objective:** The user sought assistance in completing a Java program for a digital logic simulator. The program involved a `Workspace` class with various modes (SELECT, DRAG, INSERT) for managing circuit objects. Additionally, there was a `WiringTool` class responsible for handling wiring logic.

2. **WiringTool Class:** The `WiringTool` class was extended to include methods for creating wires, extending wires, and drawing wires. The user was guided on how to integrate these methods with the `Workspace` class.

3. **Wire Class:** The `Wire` class was designed to represent wires with the ability to branch. It included methods for creating, extending, and drawing wires, as well as handling branching.

4. **Mouse Events:** The `MouseListener` and `MouseMotionListener` were set up in the `Workspace` class to handle mouse events for creating, extending, and dragging objects and wires.

5. **Direction of Wire:** The user inquired about determining the order of direction in which a wire is dragged (up and to the right or to the right and then up). A method was added to the `Wire` class to identify the order of direction based on the sequence of points.

Throughout the conversation, the code snippets were provided and explained to help the user understand and implement the required functionality in their Java program.

Week 3

Sure! Here's a summary of our conversation:

1. **Objective**: You want to add a simulation feature to your Java Swing logic gate application.

2. **Approach**:
   - Update the `CircuitObject` class to include a `simulate()` method that simulates the behavior of the object.
   - Implement specific simulation logic for each type of gate (e.g., `ANDGate`, `ORGate`) based on their functionality.
   - Add a method in the `Workspace` class to trigger simulation for all circuit objects.
   - Invoke the simulation method from the GUI when the user interacts with the "Simulate" button.

3. **Key Steps**:
   - Define simulation logic for each gate class, such as AND gate, OR gate, etc.
   - Update the `simulate()` method in the `Workspace` class to iterate through all circuit objects and trigger their simulation.
   - Implement simulation logic for wires, considering the state of input nodes.

4. **Example Code**:
   - Provided example code snippets for the `simulate()` method in `ANDGate` and `Wire` classes.
   - Demonstrated how to invoke simulation from the GUI using a "Simulate" button.

5. **Customization**:
   - You can customize the simulation logic based on your specific requirements, such as introducing delays, handling multiple simulation steps, or incorporating different gate behaviors.

6. **Conclusion**:
   - By following these steps, you'll be able to integrate a simulation feature into your Java Swing logic gate application, allowing users to observe the behavior of the circuit they've created.

Chat GPT Logs for Kyle Pickle

Week 1

Certainly! Here's a summary of the conversation so far:

1. **Project Overview**: You are creating a Digital Logic Simulator using Java Swing, similar to Logisim, that allows users to create logic circuits out of wires and gates.

2. **UML Diagram**: You provided a UML diagram outlining the architecture of the project. It includes classes like `App`, `Palette`, `Workspace`, `WiringTool`, `Simulator`, `Serializer`, `Wire`, `Register`, `LogicGateBuilder`, `LogicGate`, and `CircuitObject`. There are also interfaces `Serializable`, `Drawable`, and `Simulatable`.

3. **Maven Configuration**: You wanted to configure a `pom.xml` file to build the project with Maven, compile a single, executable JAR file, and run the `main()` function in `App.java`, which is located in the `com.github.kywillpickle` package.

4. **Initial Attempt**: I provided a `pom.xml` file that sets up a Maven project with the `groupId` `com.github.kywillpickle` and the `artifactId` `digital-logic-simulator`. However, it did not include the necessary configuration to create a single executable JAR file.

5. **Updated Maven Configuration**: I provided an updated `pom.xml` file that includes the `maven-assembly-plugin` to create a single executable JAR file with all dependencies and resources included.

Week 2

Certainly! Here's a summary of the questions you've asked and the responses I've given since the first time you asked for a summary:

1. **Creating a Digital Logic Simulator**: You asked for assistance in creating a Digital Logic Simulator using Java Swing, and I provided a brief overview of the project along with a UML diagram and a sample `pom.xml` file for Maven.

2. **CircuitObject Class**: You asked for help in creating a `CircuitObject` class that represents logic gates in the simulator, and I provided an example implementation of the class with methods for drawing, serializing, and simulating the gates.

3. **Palette and Workspace Classes**: You asked for help in creating the `Palette` and `Workspace` classes for the simulator, and I provided example implementations of the classes with methods for adding gates to the workspace and drawing the workspace.

4. **MouseMotionListener**: You asked for help in implementing a `MouseMotionListener` to redraw the canvas as the mouse hovers over the workspace, and I provided an example implementation of the listener.

5. **Selectable Objects**: You asked for help in making items on the workspace selectable, and I provided an example implementation using a `MouseListener` to detect mouse clicks on the `Workspace` `JPanel`.

6. **Zooming Workspace**: You asked for help in implementing zooming in and out of the workspace without affecting the `CircuitObjects`, and I provided an example implementation using an `AffineTransform` object to apply the zoom to the `Graphics2D` object.

7. **Shifting Reference Frame**: You asked for help in temporarily shifting the reference frame of the `draw` method in the `CircuitObject` class, and I provided an example implementation using the `translate` method of the `Graphics2D` object.

8. **Summary of Conversation**: You asked for a summary of our conversation since the last summary, and I provided a brief summary of the questions you've asked and the responses I've given.

Week 3

Here's a summary of our exchanges since the last time you asked for a summary:

1. **Creation of Source and Drain Classes**: You requested the creation of `Source` and `Drain` classes representing high/low bits in a `BitSet`, and I provided implementations extending `CircuitObject` with draw and simulate functionalities.

2. **Creation of Register Class**: You asked for a `Register` class with one input and one output, extending `CircuitObject`, and I provided an implementation with the required functionality.

These exchanges focused on creating specialized circuit components within the context of a digital logic simulator, each with specific input/output behaviors and drawing capabilities on the workspace.