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. Our "BHAG" as seen later in this doc was to be able to experiment with logic gates and to connect them in series, helping us to understand more complex circuits and logic diagrams. We believe we were able to achieve this goal and the requirements of this assignment to a decent degree, and are proud of what we have been able to put together for this project.

Literature Review/Background Study

Our project most closely resembles logic simulating programs like Logisim, which is fairly well-received as an application and also built in Java. Logisim has over 60,000 lines of code[1] however, which is safe to say is well outside of the scope of our application. 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. Programs like the one presented here can also 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.

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 more complex objects or other extraneous features depending on our intended audience. This concept of design feedback is taught in UC Davis' HCI course, but a more specific example comes from Dr. Thomas Malone's paper on "Intrinsically Motivating Instruction" cited below, in which he outlines raw data on the psychological effect this kind of feedback can have on the user.[2]

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. Kyle did his work in Eclipse and Nic in VSCode, which helped us to ensure our project works regardless of the IDE it was coded with. 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 or two per group member at a given time. We communicate on Discord and are already partners for our senior project this year, which has helped us better coordinate with each other.

Implementation Details

We did 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. As per the requirements we used Java Swing graphics for all of our GUI aspects. In addition we decided to go with BitSets to store all values across a circuit to be more flexible, even though we never ended up using more than a single bit. ChatGPT was also a vital tool to get this project completed in time, though it became progressively less helpful as our project grew.

Testing and Evaluation

We have both been extensively debugging and identifying issues to keep each other in the know while we develop our application. Before submitting we ensured that most if not all of the program was working properly, though we did not have enough time to research or implement explicit test cases.

Results and Discussion

Our decisions were greatly motivated by a desire for simplicity, ease of use, and educational value. We didn't want to overwhelm users with too many features, gates, menus, etc. These motivations can be seen with our simple GUI, our palette of common logic gates, the intuitive visual feedback during simulation, snap to grid functionality to keep the circuit clean and organized, as well as save and load features to allow users to save their work and pick up where they left off. We even went beyond our initial expectations to implement variable input / output gates. While keeping the overall features simple, we believe we have provided enough functionality for users to take advantage of in order to create more complex digital circuits. Although we have made strong progress through the 4 weeks we have developed this application, there are many ways to move forward and improve. Some features we had in mind,

that we didn't get to implementing were: Allowing users to select multiple objects to move or delete all at once, moving wires with objects when objects are moved, and adding more circuit objects such as different input sources, mux, demux, ALU, adders, clocks, etc. Ultimately, these features not being implemented came down to a lack of time, however, they were deemed less priority because they were likely too complex for our initial vision of the project.

Conclusion

Overall, we are very proud of the project that we have come up with over the course of these four weeks, and believe it would be of great use for anyone trying to learn more about digital logic circuits. This has been a great learning experience for us, teaching us many lessons about the process of software engineering. Firstly, we learned to have a vision. For us, this included drawing an initial UI, coming up with a UML diagram to visualize the structure of our software. identifying which design patterns could be of use within each structure that we came up with, and generating user stories to decide which features to prioritize and guide us through the course of the application's development. Through the user stories that we created, we recognized the value of incremental development and how it allowed us to narrow our focus, and also come together more frequently to discuss how things are going and if there are certain things we need to refactor. Lastly, the importance of git and version control was reiterated for us, as this tool allowed us to work in parallel seamlessly and also keep track of all of our changes, in the case that we needed to revert.

References and Appendices

1. Burch, Carl. "Logisim Release Statistics." Logisim: Release history, March 21, 2011. http://www.cburch.com/logisim/history-stats.html.
2. Malone, T. A. (1981). Towards a theory of intrinsically motivating instruction. *Cognitive Science,* 4, 333-369.

User Manual

Upon opening the digital logic circuit 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. In order to bring an object into the workspace, simply click on one of the gates from the palette and move your mouse into the grid. You will see the gate move along with your mouse, and when you are ready to place the gate, you can click again to set it in place. Wires can be added by dragging from the circular nodes attached to each logic gate, as well as the nodes from other wires. ***IMPORTANT*** Wires must be drawn in the direction of simulation flow (e.g. from the output of a gate to the input of another

gate). If wires are drawn the other way around (from the input of a gate to the output of another gate) the simulation will not run properly.

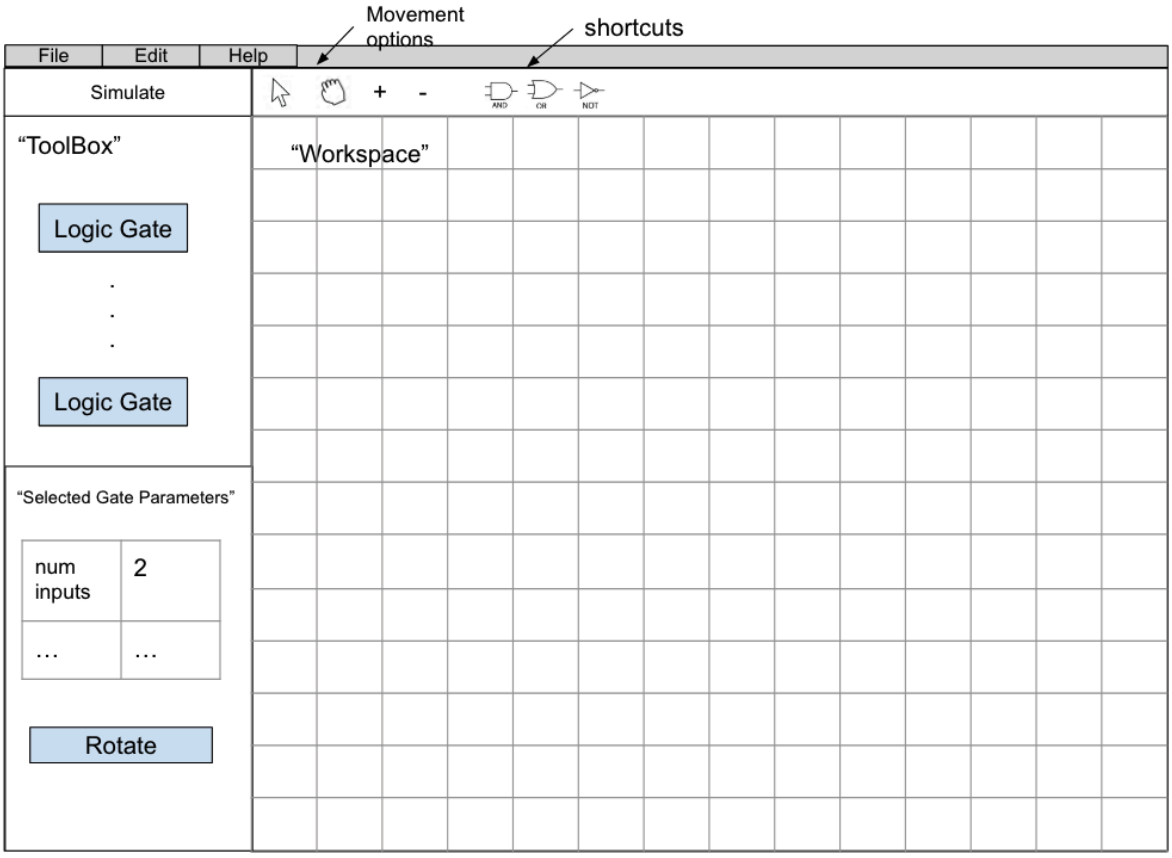
There are several options to help with movement within your workspace, such as zoom in, zoom out, select and drag. When “select” toggled, you will be able to press on different logic gates to select them, move them, delete them, or configure their parameters in the logic gate parameters panel in the bottom left. This panel will be discussed in a later section. It is also important to note that you can only create wires when in select mode. 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. This is the only functionality provided in the drag mode, so if you are not wishing to drag through the workspace, it is advisable to be in select mode.

Below the logic gate palette, 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. You will be prompted with an input and output field, corresponding to the number of inputs and outputs you wish the selected logic gate to have. Additionally, there is a rotate button, allowing you to rotate the logic gate as desired. Logic gates can also be moved with the arrow keys and rotated when pressing the r key.

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, in the top left, and it 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. Wires with current flowing through them will show up as green, otherwise, they will stay their original black color.

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 a .ser file, and load allows you to load a previously saved .ser file 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

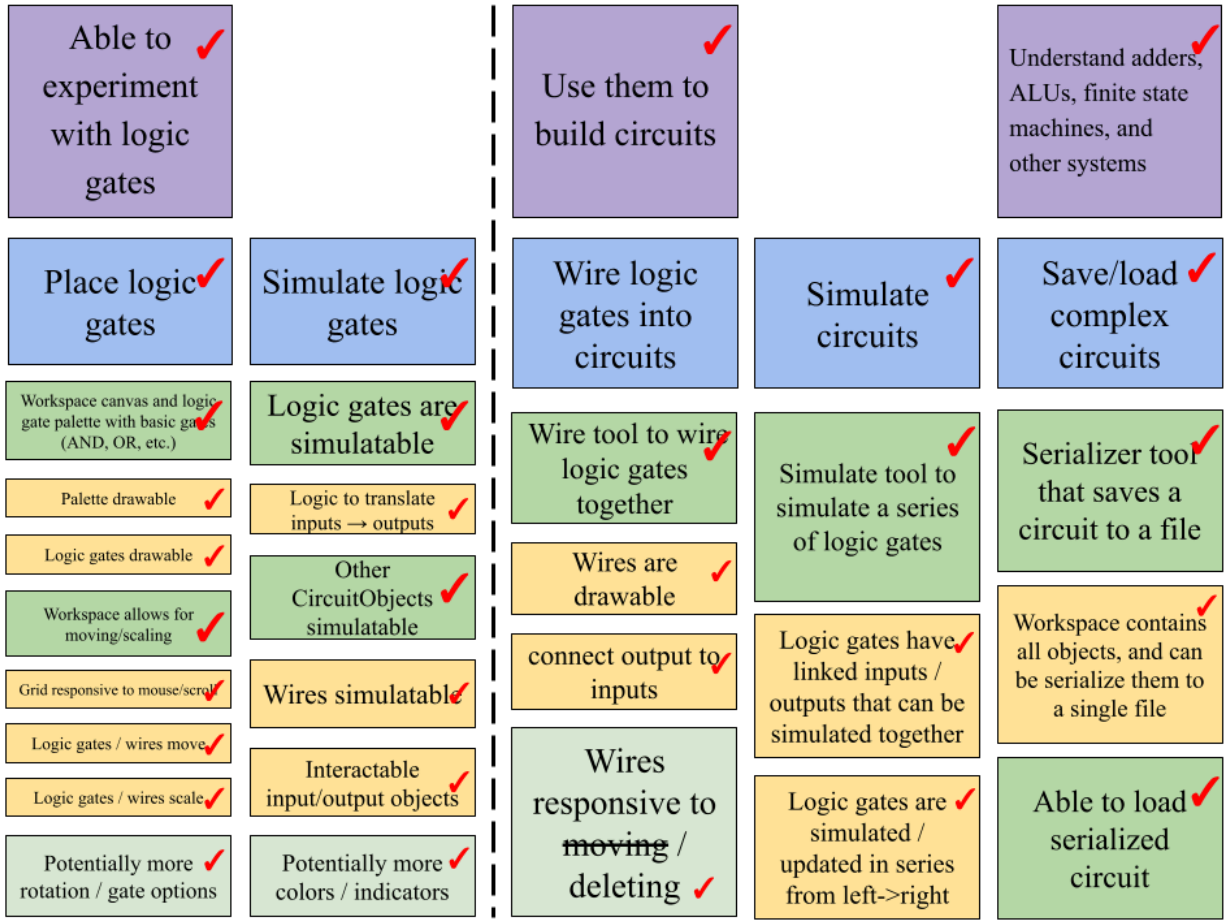


Software Design

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. Our design ended up relying heavily on the CircuitObject class, with each specific gate inheriting from it. This however worked very well for the purpose of our project and minimized the code for each gate, making it incredibly easy to add new ones. For a more detailed diagram of our class structure, refer to the class diagram below.

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 Inputs).

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.

Template

The CircuitObject class in particular, while a fully flushed-out class, is still declared abstract. Its purpose is to implement as much of the generalized functionality of a circuit object as possible, such as moving, rotating, and drawing nodes. Its derived gates and objects thus only have to implement the bare minimum, making it incredibly easy to add and work with new objects.

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.

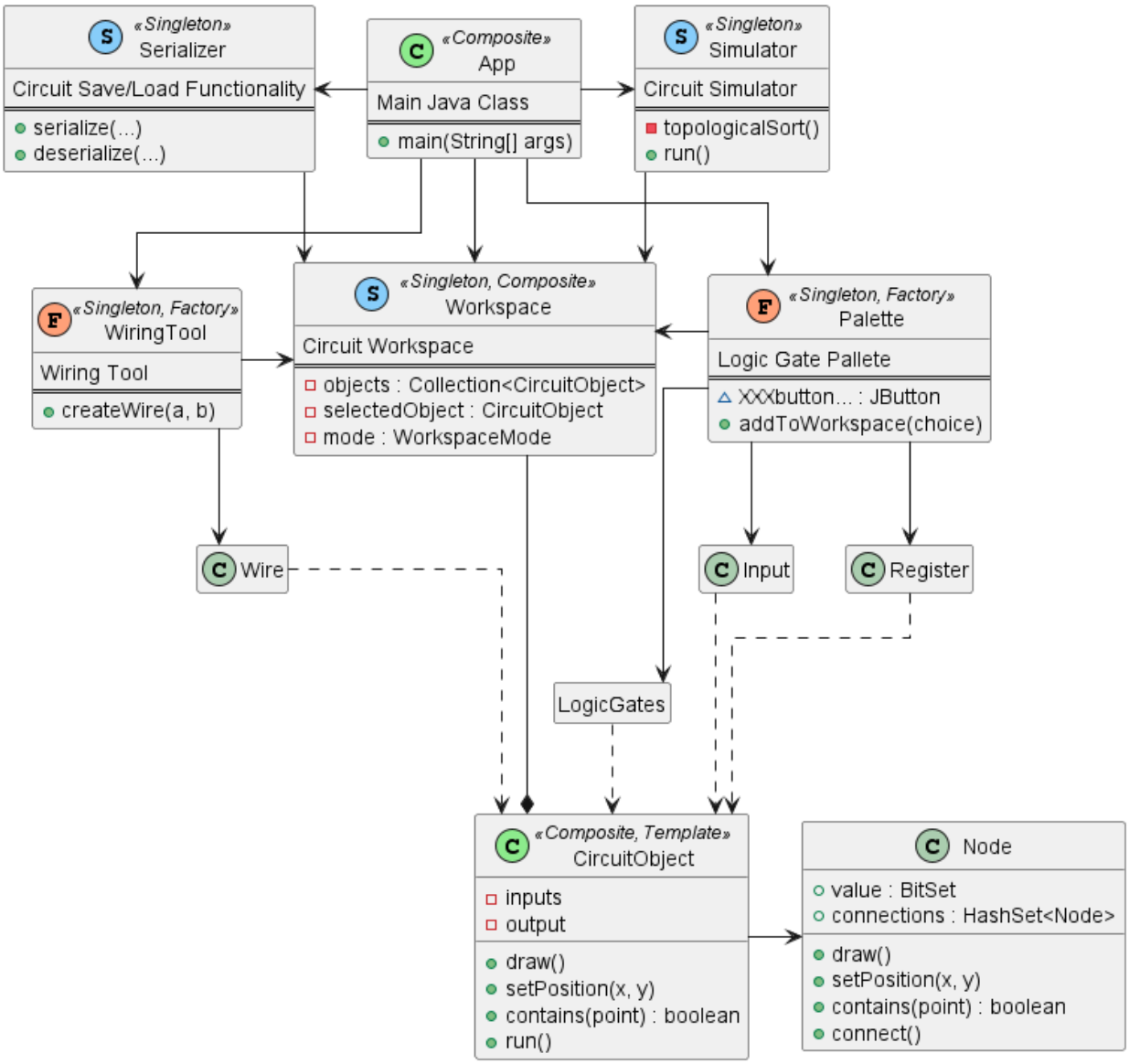
Component Descriptions

- App (Composite Pattern):
    - Main Java class responsible for initializing and running the application.
    - Stores instances of all 5 singletons, facilitating their interactions within the program.
- Palette (Singleton, Factory Patterns):
    - Represents the logic gate palette in the application.

    - Contains buttons that each are Listeners and Suppliers of select CircuitObjects
- Workspace (Singleton, Composite Patterns):
    - Represents the circuit workspace where objects are placed and connected.
    - Composed of a List of configured CircuitObjects, of which it can draw, remove, or otherwise manipulate.
- WiringTool (Singleton, Factory Patterns):
    - Represents the tool for creating and managing wiring connections.
    - Responsible for generating Wires from point a to b, automatically detecting clicked nodes and drawing from them.
- Simulator (Singleton Pattern):
    - Represents the simulation functionality for the circuit.
    - Simulates the circuit, performing a topological DFS from end to start to ensure each CircuitObject is simulated in order and loop-free.
- Serializer (Singleton Pattern):
    - Manages serialization and deserialization of circuits for saving/loading functionality.
- Wire, Input, Register (Inheritance):
    - Represents specific types of circuit objects inheriting common functionality from CircuitObject but are drawn or otherwise interacted with differently from other logic gates.
- LogicGates (Inheritance):
    - Represents the 7 logic gate classes present in the palette and in the gate package.
    - While not an explicit class, these CircuitObjects all share similar implementations that have been grouped together for organization's sake.
- CircuitObject (Template, Composite Pattern):
    - Represents a generic circuit object with inputs, outputs, drawing capabilities, position management, and simulation functionality.
    - Each CircuitObject is composed of input and output Nodes, which are connected between each other to be simulated.
    - Purposefully does a lot of heavy lifting to make it very easy to implement new inherited objects, overriding where needed (typically draw() and run()).
- Node (Composite Pattern):
    - Represents a connection point for wires and connections between circuit objects.
    - Compared and linked together through its parent CircuitObject to form a connected circuit.

Diagrams

The following is a rough class diagram of the above objects rendered with PlantUML (https://www.plantuml.com/plantuml/uml/).



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

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

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

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.

Week 4

1. **Serialization and Deserialization Issues:**
- You encountered a `java.io.NotSerializableException` when attempting to serialize the `Workspace` class, likely due to non-serializable fields or references.
- We discussed different approaches to address this issue, including making non-serializable fields transient, implementing the Serializable interface, or handling the non-serializable fields separately.

2. **Loading Circuit Objects:**
- You asked about how to reload the list of circuit objects after deserialization in the `Workspace` class.
- We discussed implementing a `loadWorkspace()` method that reads the serialized data from a file, clears the existing list of circuit objects, adds the loaded objects, and then performs any necessary initialization steps.

3. **Saving Circuit Objects with Specific Extension:**
- You wanted to restrict saving to files with a specific extension, such as ".ser".
- We added a file filter to the `JFileChooser` dialog to ensure that the user can only save files with the ".ser" extension.

4. **User Interaction for Saving Files:**
- You asked about allowing the user to specify the filename and location when saving a file.

- We implemented a `saveCircuit()` method that opens a `JFileChooser` dialog for the user to choose the file location and name, and then delegates the save operation to the `Workspace` class.

5. **Additional Functionality:**
   - We discussed the possibility of adding custom initialization steps after loading circuit objects, such as relinking nodes or performing other necessary setup.

Overall, we covered serialization and deserialization issues, file saving with specific extensions, user interaction for file saving, and additional functionality for handling circuit objects.

Chat GPT Logs for Kyle Pickle

Week 1

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

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

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.

Week 4

1. **Class Design and Implementation**:
   - Designed and implemented classes such as `Source`, `Drain`, and `Register` extending `CircuitObject` to represent digital logic components in the simulation.
   - Created a `Node` class to manage connections between circuit objects, adhering to the Composite pattern.

2. **User Interface and Interaction**:
   - Integrated Swing components like buttons and text fields for user interaction, enhancing the application's user interface.
   - Explored techniques for drawing rotated objects in Swing and managing coordinate systems for graphical representation.

3. **Input Handling and Global Key Events**:
   - Implemented key event handling in the workspace and main JFrame to detect global key inputs, ensuring that key events are processed regardless of component focus.

4. **Design Patterns and Architecture**:
   - Updated the UML diagram to reflect design patterns such as Singleton, Factory, Composite, and Inheritance, showcasing a well-structured architecture for the application.

5. **UI Enhancements and User Experience**:
   - Improved UI components by adding icons to buttons and controlling button layout, enhancing the overall user experience of the Digital Logic Simulator.

6. **Simulation Logic and Object Order**:
   - Discussed strategies for simulating circuit objects in a specific order based on their connections, ensuring correct functionality and behavior during simulation.

These discussions and implementations have contributed to the development of a robust Digital Logic Simulator application with a focus on design patterns, user interface design, input handling, and simulation logic.