Project Report

Introduction

The digital logic simulator project aims to create a user-friendly, interactive environment for building and testing digital circuits.

Background

Logisim, a key reference for this project, offered valuable lessons in user experience and functionality. The goal of the project is to create a simpler but more intuitive design compared to Logisim. Although Logisim is very feature-rich and has a great drag-and-drop interface, some features like the wiring can be hard to use. In addition, the overwhelming number of options on the lefthand side can make it less intuitive when trying to create a very simple digital logic circuit design.

In contrast, this project is very simple and intuitive, with almost every option available to the user visible and easy to understand at a glance at the moment of starting up the program.

Methodology

The project adopted an agile development methodology, with iterative testing and refinement. Java Swing was used for the graphical user interface, ensuring cross-platform compatibility and ease of use.

First, goals were brainstormed and set for each week to align with each of the weekly ECS160 project submissions.

Implementation Details

The simulator is being implemented in phases, starting with a large-scale skeleton overview of the project. Each item is being added in incremental steps. For example, the grid was first being created statically without zooming and was expanded to include this feature. Likewise

for many of the components in the project like the `Logic Gates`, first one gate like the `AndGate` was worked on and then each of the remaining gates were implemented afterwards based on the first gate.

Testing and Evaluation

Testing was done incrementally, parallel to the development of the project. As features were added one-by-one, the implementation of each part was tested by running the program or by using debug print statements.

Results and Discussion

The first week took a lot of trial and error learning about Java Swing, brainstorming the architecture of the project, and working with ChatGPT effectively. Most of the work done was experimenting with drawing objects and the interface of the program.

In the second week, much of the project was rewritten to manually handle the transition of drag and drop objects from the `PalettePanel` into the `GridPanel` instead of using the `TransferHandler` interface and multiple JPanels in a Border style layout. This allowed for smoother transitions but required the creation of a `MainPanel` class that interfaced between the `PalettePanel` and `GridPanel`.

In the third week, the gate display was updated to use images instead of drawing using Java Swing graphics. In addition, resizing functionality was added to the Grid.

In the final week, it became clear that there needed to be multiple mouse modes to handle moving components versus creating wires and deleting components.

Conclusion

Since this was the first GUI project I had ever undertaken, I had to undergo a lot of experimentation to see what worked in terms of user interface (UI) and user experience (UX).

If the project were to be developed further, the first large feature to be added would be panning the grid. In addition, it would be useful to have a `Select` mode and have the functionality to select multiple components and wires and move them all at once. There are a

lot of buggy parts of the program such as when right-click dragging objects that could also be ironed out.

References and Appendices

ChatGPT

This was my first time working with ChatGPT on a larger project. At the beginning of the project, I relied more on ChatGPT to come up with broader ideas and code to set up the project. At the start of the project, I asked about typical Java Swing project structures and how to set up a GUI, as can be seen in `/ChatGPT/02-22.pdf` and `/ChatGPT/03-01.pdf`.

However, it soon became apparent that ChatGPT was best utilized to solve small-scale issues, like a more-specifically tailored search-engine. At the tail-end of the project, I asked more well-defined questions with clear answers, like whether or not a Java `HashSet` could hold `Point` objects and correcly do `.equals()` comparisons between objects when calling `HashSet.contains()`. This can be seen in `/ChatGPT/03-21.html`.

Other references used in the project

- Logisim software
- Java Swing documentation
- Logic Gate images taken from https://www.categories.acsl.org/wiki/index.php?title=Digital_Electronics
- Mouse button image taken from https://en.m.wikipedia.org/wiki/File:Mouse-cursor-hand-pointer.svg
- Add button image taken from https://www.cleanpng.com/png-plus-and-minus-signs-clip-art-3214188/
- Images made transparent using https://www.remove.bg/

User Manual

Setup

Launch the Application by running the App.java file `/src/main/java/logicsim/App.java`.

Functionality

Opening and Saving Circuits

Click the top menu option `File` and then the `Save` or `Open` menu options. Alternatively, use the keyboard shortcuts `Ctrl/Cmd + S` and `Ctrl/Cmd + O` for `Save` or `Open`, respectively.

Only `.xml` files that were created using this program are supported for opening.

Example circuits can be found im the `exampleCircuits` folder at the root of the project.

Adding and Moving Components

Clicking and dragging a component allows you to add the Logic Gate to the Grid on the right-hand side. Each component will snap to the grid when dragging, and a visual shows the current position of the dragged object.
Objects placed on the grid are able to be moved with another drag and drop operation.

Adding Wires

Click the `+ Wire` button at the top left of the screen or press the keyboard shortcut `W` to switch to the wire-adding mode. When the mouse hovers over the grid panel, it will now show a preview of a wire that can be added by clicking the left mouse button.

Remove Components and Wires

Click the `X Delete` button at the top left of the screen to switch to the delete mode. When the mouse hovers over any previously placed wire or component on the grid, the hovered

item is highlighted in red. Clicking the left mouse button will remove the item from the grid.

When multiple items are highlighted at the same time, the deletion prioritizes `Wires` then `InputOutputComponents` then `Gates`.

Toggle Input 0-1

The Input component can be switched between `0` and `1` by switching to the `Move` mode and then highlighting and clicking the right mouse button on the input component you want to toggle.

Currently, the right click functionality is slightly buggy, especially when accidentally performing a `right click and drag` operation.

Zoom the Grid

Use the mouse wheel to zoom in and out on the grid.

Design Manual

Architecture Overview

The application is structured around the main `MainPanel` class, which extends `JPanel`. It orchestrates the overall layout and interactions between the `PalettePanel` and `GridPanel`.

- `App`: Serves as the main entry point for the application.
- `MenuBar`: Contains menu items for `Save` and `Open` operations
- `MainPanel`: Handles mouse events and interactions between the PalettePanel and GridPanel
- `PalettePanel`: Displays useable LogicGates and information.
- `PaletteComponent`: Represents a draggable component in the `PalettePanel` with properties such as `logicGate` and `bounds`. It includes a `draw()` method for rendering.
- `GridPanel`: Handles grid components and wires using relative coordinate system
- `GridLogicHandler`: Handles the logic related to the grid, including interactions with logic gates and wires.
- `LogicGate`: An abstract class extended by specific gate types like `ANDGate`. It provides essential methods like `draw()`, `contains()`, and `output()` to handle logic gate functionalities.
- `WireComponent`: Has `start` and `end` points and can be drawn to the grid.

Each of these components works together to provide a comprehensive digital logic simulation environment.

Design Patterns

Singleton

The `PalettePanel`, `GridPanel`, `GridImporter`, and `GridExporter` all utilize the Singleton pattern.
There are no cases with the application that there would need to be more than a single

version of these classes. For example when the `GridExporter` needs to export the current grid layout to an XML file, export is called on the single instance of `GridPanel`. The Singleton allows for easy interfacing and synchronization between the classes.

Strategy

The `MouseModes` for the project made it possible to swap between the different ways to interact using the mouse quickly and easily. This is similar to the `Strategy` design pattern since the program can switch between different algorithms at runtime to interface with the program.

Component Descriptions

App

At the top of the architectural hierarchy is the App class. This initializes the window with MainPanel and MenuBar objects.

MenuBar

The MenuBar contains commands for saving and loading circuits. There are also menu options for exiting the application, and menu options for an `About` page and `Documentation` page with a hyperlink.

MainPanel

The MainPanel contains the logic for handling mouse operations (may be refactored out to a MouseHandler class). This class manages the interaction between the `PalettePanel` and `GridPanel`, such as drag and dropping new logic gate items from the `PalettePanel` into the `GridPanel`.

PalettePanel

The `PalettePanel` displays `PaletteComponent`s which each house a `LogicGate` component. Each of the default gates are shown on the lefthand size of the application. When a component is hovered, the component colors the background of the component gray to

make it clear to the user.

GridPanel

The GridPanel displays a grid and the current view of the logic gates and wires. In future updates, a `GridLogicHandler` class will be added that calculates outcomes based on the circuit on the grid.
When dragging a component around the GridPanel, components snap to the grid.

GridLogicHandler

Whenever some component or wire is added or removed from the grid, the `checkLogic()` function is called to check the validity of the circuit.

First, the components are checked for overlaps in which case the component is highlighted in red and the check ends early.

When no components are overlapping in an illegal way, the grid is searched greedily starting from the `InputComponents` like a Depth-First-Search (DFS) which is implemented using a Stack data structure. For circuits such as Flip-Flops, the input for part of a gate is dependent with another gate, creating a circular dependency. To tackle this issue, the logicHandler lazily determines the output of some gate with just a single input if it possible to do so. One example would be an `AndGate`. If one input of an `AndGate` is False, it is possible to determine that the `AndGate` must output False.

LogicGate

Logic Gates are implemented as an abstract class which is extended by each gate such as ANDGate, ORGate, and XORGate. Each logic gate have associated image files in `resources/gates`.

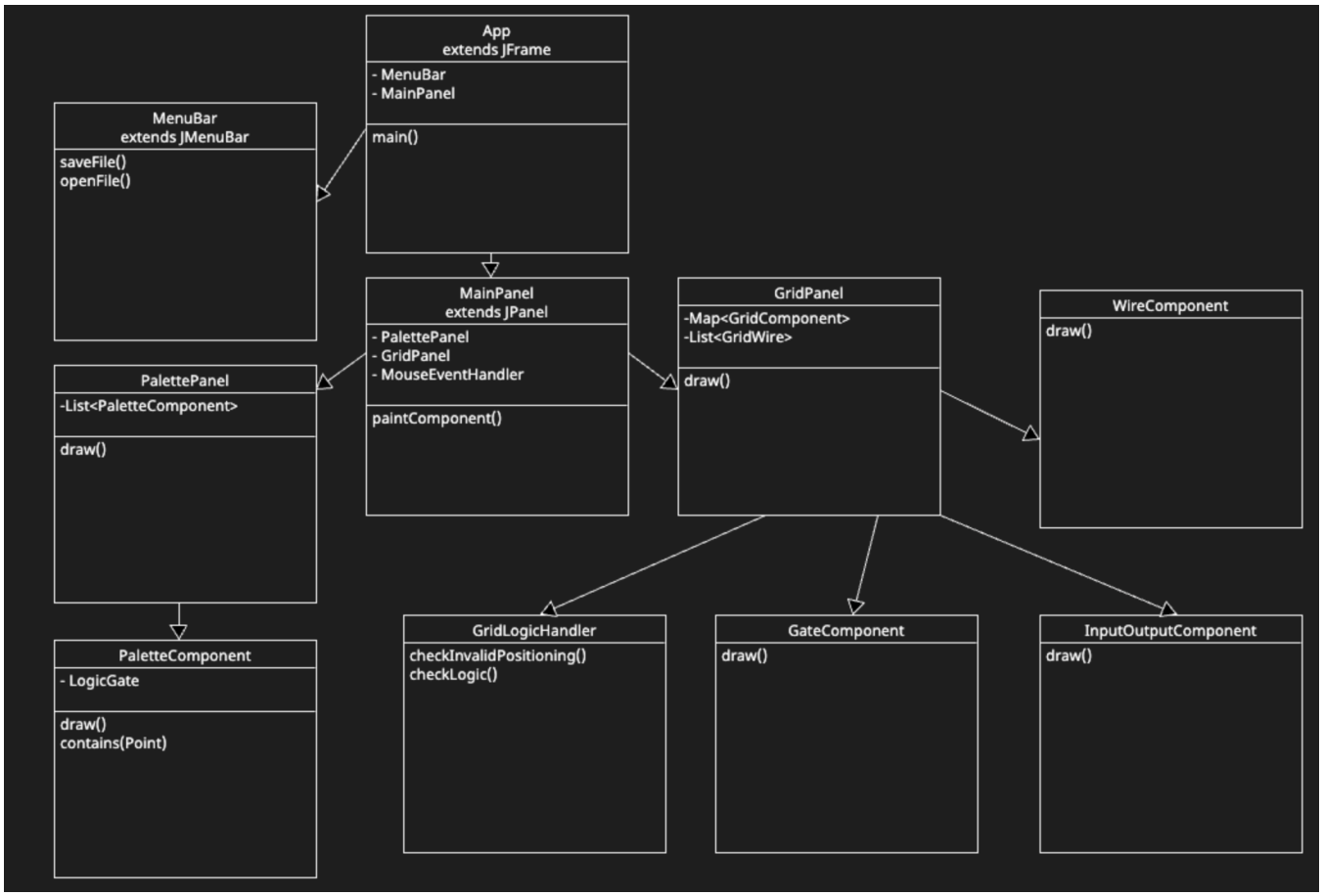
GateComponent

Instead of having the `LogicGate` interface directly with the `GridPanel`, the functions and parameters only necessary with the GridPanel were extended using this GateComponent class.

WireComponent

When a `WireComponent` is initialized, the points are sorted by lowest x value then lowest y value to make it easier to check for duplicate wires.

Diagrams



Standards and Conventions

The project adheres to Java's camelCase naming convention, with classes beginning with uppercase letters, methods as verbs, and objects as nouns.