Tech Titans

Boolean Logic Simulator
Software Architecture Document

Version 1.5

Revision History

Date	Version	Description	Author
11/04/2024	1.0	Initial Document Work	<name>
13/04/2024	1.1	Quality	Theodora
13/4/2024	1.2	Interface details	Justin
13/04/2024	1.3	Acronyms, references, and overview	Cole Charpentier
13/04/2024	1.4	Section 5	Caden LeCluyse
13/04/2024	1.5	Section 2 and 3	Ceres Botkin
14/04/2024	1.6	Section 1 through final	Alexander Phibbs

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

This document gives the overarching architecture of our project, specifically this is being made for our instructors to clearly see we have put thought and effort before the coding starts to map out a good framework to see once the project is created if we follow through. The best way to use this document is to check for sound logic across the board and then see how we follow through. If any of our code differs this is not a problem but should be well documented why each change was made to show growth in this learning process.

1.2 Scope

This project document will cover our general outline or architecture of our project. This includes different definitions that we will use and be assumed to know in our project, this also includes the specific architecture requirements given to us by our instructor that we will detail in the constraints.

1.3 Definitions, Acronyms, and Abbreviations

CLI: Command line interface

C++: A high-level, general-purpose programming language. This is the programming language our program will be written in.

AND (&): A logical operator which returns true if both operands are true.

OR (|): A logical operator which returns true if at least one operator is true.

NOT (!): A logical operator which inverts the truth value of its operand.

NAND (@): A logical operator which returns true if both operands are false.

XOR ($): A logical operator which returns true if exactly one operand is true.

See the project glossary.

1.4 References

We did not need any additional documents mentioned in this specific document

1.5 Overview

The rest of the Software Architecture Document contains several sections, including the architectural representation, architectural goals and constraints, logical view, interface description, and quality. These sections describe various architectural decisions made during the development process for the Boolean Logic Simulator. Additionally, the quality section describes how the software architecture contributes to all capabilities of the system other than functionality.

2. Architectural Representation

We use a layered architecture to represent the data in the system and how it is processed. We find this necessary because it stresses how the data is manipulated into a final result. The first layer is the interface layer which gets the data as input, and then transforms the data into a logical form on the logical layer which makes it easier to understand and be evaluated. After evaluation we are given a final evaluation which can be returned to the user. Each of these layers make up a single module we will use.

3. Architectural Goals and Constraints

Our objectives include making a clear distinction between layers and having a good method of getting data

between each part. We aim for high cohesion inside each layer and low coupling between each layer. We will achieve this by only passing one set of data between each layer. In addition, the layered approach emphasizes modularity given its ability to transform data of other types if we need.

One of our constraints is that it must be able to run in the terminal. This will ensure that the result can be piped if needed, which could lead to higher forms of modularity. In addition, we aim to write in C++. This can also increase modularity because we could import each layer into another C++ program if needed. We will implement each layer at a time and move on when we have the desired output. We will also follow a core design document like our SRS.

4. Use-Case View

[This section lists use cases or scenarios from the use-case model if they represent some significant, central functionality of the final system, or if they have a large architectural coverage—they exercise many architectural elements or if they stress or illustrate a specific, delicate point of the architecture.]

4.1 Use-Case Realizations

[This section illustrates how the software actually works by giving a few selected use-case (or scenario) realizations, and explains how the various design model elements contribute to their functionality. If a Use-Case Realization Document is available, refer to it in this section.]

5. Logical View

5.1 Overview

This section will break down the overall design model of the program from a hierarchical perspective

5.2 Architecturally Significant Design Modules or Packages

5.2.1 Interface layer:

- A standalone class will not be needed for this program when handing the user interface. All information seen by the user will simply be handled in main.cpp

5.2.2 Logic Layer:

Abstract Syntax Node Class:

- This class will represent the boolean operation between two operands, giving a result

Expression Parser Class:

- A class that will deal with each of the boolean operators, as well as parentheses
- It will use the abstract syntax node class to fully evaluate the expression
- The class will throw an error if such an error is detected

6. Interface Description

The arithmetic expression evaluator will be a command-line interface (CLI). Here's a description of the major entity interfaces:

User Input Format:

Users will interact with the program by entering arithmetic expressions containing operators and numeric constants.

If the user enters an invalid expression or encounters an error during evaluation (e.g., division by zero), the program will provide clear and informative error messages.
Error messages will guide the user on how to correct the input or address the issue encountered.

7. Size and Performance

[A description of the major dimensioning characteristics of the software that impact the architecture, as

well as the target performance constraints.]

8. Quality

Extensibility

- Modular Design: Organize software into discrete modules with clear interfaces, all capable of easy change without impacting the existing system functionality and features

Reliability

- This justifies that the architecture can be designed in a way that contains redundant components and failover mechanisms; these will assure that the operation can continue even if failure of hardware or software occurs

Portability

- For example, hardware abstraction layers or virtual machines allow software to run without modification on different hardware platforms

Security

- Strong Encryption and Authentication Protocols: Strong encryption techniques along with authentication protocols would be pierced into the architecture so that data is protected and not liable to any sort of breaches or manipulations.

Privacy

- Data Encryption: Ensuring data is stored encrypted at rest and during data in motion from any unauthorized access, which is the whole point with personal and sensitive data.