Kyler_Jack_Cyn_Malek_Kaleb_Jackson Co

The Logic Calculator
Software Architecture Document

Version 1.0

Revision History

Date	Version	Description	Author
4/10/2024	1.0	Partial of 3 with suggestions of architectural goals and constraints and partials of 2 architectural representation	Kyler Luong
4/12/2024	1.0	Outline in 5.1 the program design layers and architecture, Describe in 8 how the program is reliable and more functionable.	Kaleb Howard
4/13/2024	1.0	Complete 1.1 Purpose, 1.2 Scope, 1.3 Definitions, Acronyms, and Abbreviation, 1.4 References, 1.5 Overview	Cyn Tran
4/12/2024	1.0	Complete, section 5.2 Architecturally Significant Design Packages, 4 Use-Case View, and 6. Interface Description	Jackson, Jack, Malek

Table of Contents

Software Architecture Document

1. Introduction

The introduction section of the Software Architecture Document offers a preview of the entire document. Covering the document's purpose, scope, definitions, acronyms and abbreviations, references, and an overview of the project Software Architecture Document.

1.1 Purpose

This document serves as a comprehensive guide to the inner workings of the project's system. It describes a detailed look from various angles to showcase different aspects of how the program is put together and highlights the major decisions the team made while building and shaping the system.

This section emphasizes the role of the Software Architecture Document within our project documentation. Providing an overview of the internal structure, enabling stakeholders to understand its contents better. Additionally, the documentation identifies the team members and their responsibilities in the implementation of the system.

1.2 Scope

The Software Architecture Document defines the boundaries of the project's software system application, outlining the areas the software system needs to meet. Provides details on the software architecture's build, structure and function. Depicts the influence of software architecture decisions on the development, technology selection, and system design, ensuring consistency and coherence across different aspects of the project.

1.3 Definitions, Acronyms, and Abbreviations

1. Architectural Representation: The method used to illustrate the design and structure of a software system
2. API: An abbreviation for Application Programming Interface, defining the methods and protocols for interaction between software components.
3. CPU: Abbreviation for Central Processing Unit, the primary hardware component that executes instructions and processes data.
4. HTML: An acronym for Hypertext Markup Language, the standard language for creating web pages and applications.
5. CSS: Abbreviation for Cascading Style Sheets, the language used to describe the presentation of HTML documents.
6. UI: Short for User Interface, it's the interface through which users interact with the software.
7. GUI: Abbreviation for Graphical User Interface, an interface allows users to interact with devices through graphical elements.
8. HTTP: Abbreviation for Hypertext Transfer Protocol, the protocol used for transmitting hypertext documents over the Internet.

1.4 References

[IEEE830-1998]: IEEE Recommended Practice for Software Requirements Specifications, IEEE Computer

Society, 1998.

EECS 348 Project Description, Dr Hossein Saiedian, 00-2024-EECS348-project-description.pdf

Document exists in canvas projects files. Already pre-created and referenced in this document.

1.5 Overview

This documentation offers a concise and comprehensive summary of the Project's Software Architecture contents. Outlining the software structure and key components, providing stakeholders with a roadmap for navigating through the process of building the software architecture. This documentation serves as a preview of the insights and analyses that stakeholders can expect to find while working on the project including: Architectural Representation, Architecture Goals and Constraints, Use-case View, Logic View, User Interface Description and Quality.

2. Architectural Representation

We will create a prototype of our program to demonstrate our program to stakeholders and the progress it will make from the prototypes to the final. There will also be well formatted comments throughout making our program to ensure design decisions and how the implementation works

The Boolean Logic Simulator software is based on a layered architectural representation. The logic engine, which is crucial to the architecture, parses and evaluates Boolean expressions and produces the relevant truth values. Layers for the user interface, data administration, and system integration around the logic engine. The seamless interaction of these layers gives users a simple and intuitive experience.

- **Architectural Views:**
  - **Logical View:**
    - This view illustrates the logical structure of the Boolean Logic Simulator system, focusing on the decomposition of the software into manageable modules and components tailored for Boolean logic operations.
    - Model elements include classes representing logical entities such as Boolean expressions, operators, and evaluation algorithms. Additionally, interfaces defining communication protocols between modules and packages encapsulating related functionality are included.
    -
  - **Process View:**
    - The process view elucidates the dynamic behavior of the Boolean Logic Simulator system during runtime, depicting how different processes and components interact to perform logical operations.
    - Model elements encompass processes representing execution units, and components encapsulating specific functionalities such as expression parsing and evaluation.
  - **Use-Case View:**
    - The use-case view provides an overview of the system's functionality from the perspective of its users, highlighting the interactions between users and the system to accomplish specific tasks.
    - Model elements include primary and alternative use cases, actors representing system users, and the relationships between use cases and actors.
  - **Development View:**

- The development view focuses on the software development aspects of the Boolean Logic Simulator system, detailing the organization of source code, build configurations, and development environments.
- Model elements encompass modules representing logical units of code organization, dependencies defining relationships between modules, and development tools facilitating the software development process.

3. Architectural Goals and Constraints

For our product, it will be easily distributable as it is easily downloadable and also reusable as its function is used as a calculator. Our product will be downloaded to users' machines so they can use it whenever, and wherever their machine travels.

- **Goals:**
  - **Modularity:** The architecture should allow for the encapsulation of distinct functionalities within separate modules which enhances maintainability and facilitates future enhancements.
  - **Usability:** The user interface layer should be designed to be intuitive and user-friendly, enabling users to input Boolean expressions easily and interpret the results effectively. Clear and concise feedback mechanisms should be incorporated to guide users through the simulation process.
  - **Performance:** The logic engine should ensure efficient parsing and evaluation of Boolean expressions even for complex inputs.
  - **Reliability:** The software should be reliable and robust, capable of handling errors gracefully and providing accurate results under varying conditions.
- **Constraints:**
  - **Compatibility:** The software must be compatible with most Operating Systems as well as popular web browsers such as Google Chrome, Mozilla Firefox, and Microsoft Edge, ensuring broad accessibility across different platforms.
  - **Resource Limitations:** The software should be designed to operate within resource constraints such as memory limitations and processing power, particularly for deployment in resource-constrained environments such as low-spec hardware.
  - **Time Constraint:** The architectural goals should be achieved within the duration of a semester, necessitating efficient planning, design, and implementation processes. Adherence to project timelines is crucial for timely completion and successful delivery.

4. Use-Case View:

The use-case view of the Boolean Logic Simulator software encompasses key functionalities such as inputting Boolean expressions, simulating their behavior, and visualizing the results. Each use case is realized through interactions between the user interface layer and the logic engine, facilitating a seamless user experience. Use cases include inputting expressions, displaying truth tables, and providing feedback on input validity.

4.1 Use-Case Realizations

Inputting Boolean Expressions:

- Description: Users interact with the user interface layer to input Boolean expressions into the system.

- Realization: The user interface module provides input fields where users can enter Boolean expressions using standard logical operators (AND, OR, NOT) and parentheses for grouping. Upon submission, the expression is passed to the logic engine for parsing and evaluation.

Displaying Truth Tables:

- Description: Users request the generation of truth tables based on the input Boolean expressions.
- Realization: Upon receiving a request from the user interface layer, the logic engine generates a truth table for the input expression. The truth table is then formatted and returned to the user interface for display.

Providing Feedback on Input Validity:

- Description: Users receive feedback on the validity of their input Boolean expressions.
- Realization: The user interface module includes validation mechanisms to check the syntax and structure of input expressions. If an expression is invalid, appropriate error messages are displayed to the user, guiding them to correct their input. Validation is performed both client-side and server-side to ensure data integrity.

Simulating Boolean Logic Operations:

- Description: Users simulate the behavior of Boolean logic operations based on the provided expressions.
- Realization: Upon user request, the logic engine evaluates the input Boolean expressions and generates corresponding truth values (true or false) for all possible combinations of input variables. The results are then formatted and displayed to the user through the user interface layer, allowing users to visualize the behavior of the expressions.

5. Logical View

This section illustrates the important part of the architecture design, its subsystems and packages. And take a look at the different classes and utilities inside. Introducing key classes and discussing their function, along with their important relationships, operations, and attributes.

5.1 Overview

User Interface Layer:

- Handles user input of boolean expressions and truth values.
- Displays the final truth value output.

Logic Layer:

- Parses boolean expressions.
- Evaluates the expressions.
- Handles truth value input.

Error Handling Layer:

- Manages error handling for invalid expressions, missing parentheses, etc.

5.2 Architecturally Significant Design Modules or Packages

User Interface Layer:

Responsible for handling user input of boolean expressions, truth values and displays the final truth value output.

Classes:

1. **UserInputHandler:** Manages user boolean expressions input.

Responsibilities: Validates and processes user input.

Operations: getInput(), validateInput(), processInput().
Attributes: userInput (string).

2. **TruthValueDisplay**: Displays the final value output.
Responsibilities: Formats and presents truth values to the user.
Operations: displayTruthValue().
Attributes: truthValue (boolean).

Logic Layer:
Parse boolean expressions, evaluating them, and managing truth value input.

Classes:
1. ExpressionParser: Parses boolean expressions.
Responsibilities: Analyzes and breaks down boolean expressions into components.
Operations: parseExpression().
Attributes: expression (string).

2. ExpressionEvaluator: Evaluates boolean expressions.
Responsibilities: Computes the truth value of boolean expressions.
Operations: evaluateExpression().
Attributes: expression (string), truthValue (boolean).

Error Handling Layer:
Manages error handling for various scenarios such as invalid expressions and missing parentheses. It ensures smooth error management throughout the system.

Classes:
1. ErrorHandler: Manages error handling for the system.
Responsibilities: Detects and handles errors gracefully.
Operations: handleError().
Attributes: errorCode (int), errorMessage (string).

6. Interface Description

The interface of the Boolean Logic Simulator encompasses both the user interface (UI) and the software interfaces necessary for system functionality.

User Interface (UI):

The user interface provides an intuitive platform for users to interact with the Boolean Logic Simulator. It includes the following components:

- **Input Field:** Users can input Boolean expressions using a text input field. The input field supports

standard logical operators and parentheses for grouping.

- **Evaluation Button:** A button allows users to initiate the evaluation of the entered Boolean expression.
- **Output Display:** The system displays the evaluated expression result in a designated output area.
- **Error Messages:** In case of invalid input, the system provides informative error messages to guide users in correcting their input.

Software Interfaces:

The software interfaces enable communication between different components of the system, facilitating its operation. These interfaces include:

- **Logic Engine Interface:** The logic engine interface defines methods for parsing and evaluating Boolean expressions..
- **Data Management Interface:** The data management interface facilitates the storage and retrieval of user inputs and evaluated results.
- **Error Handling Interface:** The error handling interface provides methods for detecting and reporting errors encountered during expression evaluation. It allows the system to generate and display informative error messages to users when input validation fails.

8. Quality

The software architecture includes code with the sole purpose to catch errors, determine what error is occurring, and report that to the user. This does not directly help with the functionality of the program, but makes for easier debugging, error handling, and a smoother experience for the intended user of the product.

The code will be built in a pattern that has multiple, almost completely separate subsections that can each be tested on their functionality. This enables easier updating of specific functionality, easier error finding, and also allows us to add or update the code how we like at any point without the concern of having to completely rewrite the program.