Team8 Co

Calc.co
Software Architecture Document

Version <1.1>

Revision History

Date	Version	Description	Author
10/Nov/2024	1.0	Completed the first version of the architecture design document	Kai
10/Dec/2024	1.1	Updated the logical view, Architecture representation, and Goals and Constraints	Abdulaziz

Table of Contents

Software Architecture Document

1. Introduction

This document provides a comprehensive overview of the software architecture, detailing the design and implementation strategies that ensure efficient and reliable expression evaluation. It includes key architectural decisions, significant components, and quality attributes. This serves as a reference for developers, stakeholders, and maintainers to understand and enhance the system as needed.

1.1 Purpose

This document provides a comprehensive architectural overview of the Calc.co system, a command-line calculator written in C++. It is intended to detail significant architectural decisions, including recursive descent parsing for arithmetic operations, and the design choices that ensure compliance with PEMDAS (Parentheses, Exponents, Multiplication and Division, Addition and Subtraction). This document targets developers, system architects, and stakeholders, offering insights into system design and functionality.

1.2 Scope

This document pertains to **Calc.Co**, specifically outlining the parsing logic, calculation modules, operator classes, sorting logic, and the user interface (UI) structure. It impacts both the development and maintenance phases, as it provides guidelines and a structural understanding necessary for development, testing, and potential future enhancements.

1.3 Definitions, Acronyms, and Abbreviations

This section provides a glossary of terms and acronyms used in the document:

Command line interface(CLI): A text-based user interface that allows users to interact with a computer or software's components.

PEMDAS: The order of operation precedence in arithmetics - **P**arentheses, **E**xponents, **M**ultiplication, **D**ivision, **A**ddition, **S**ubtraction

Token: a singular number or operator

Grammar Rules: A set of rules describing how **Tokens** can combine to form valid expressions that follow the hierarchical structure of **PEMDAS**.

Recursive Decent Parsing (RDP): A top-down parsing technique that uses recursion and nodes to process the input based on **grammar rules**.

1.4 References

Iteration Plans
Date: 26/09/2024
Source: Iteration Plan

Vision Document
Date: 26/09/2024
Source: Calc.Co Vision Plan

Glossary
Date: 09/26/2024
Source: Glossary

GitHub
Date: 09/19/2024
Source: https://github.com/BigIronDestroyer/Calc.Co

1.5 Overview

This Software Architecture Document is organized as follows:

- **Section 2: Architectural Representation** – Outlines the models and diagrams representing the system.
- **Section 3: Architectural Goals and Constraints** – Details the requirements and constraints impacting the system design.
- **Section 4: Logical View** – Describes the system's logical decomposition into modules, classes, and interactions.
- **Section 5: Interface Description** – Defines the user interface and system interaction points.
- **Section 6: Quality** – Highlights non-functional qualities like extensibility, reliability, and security.

The document provides a top-down analysis of the architecture to ensure comprehensive understanding and effective use by all stakeholders involved in the project.

2. Architectural Representation

The architecture for **Calc.co** is designed using structural and behavioral models to provide a comprehensive understanding of the system's functionality and data flow. The system is composed of the following main components:

- **Tokenizer Module:** This module analyzes user input and organizes it based on whether a given character is a NUMBER, OPERATOR, or PARENTHESES. This process ensures that the input has no invalid characters and makes it easy for the parser to access the input.
- **Parse Module**: This module is responsible for analyzing all the tokens and enforcing the PEMDAS order of operations. This process ensures that the user input is a valid expression and that the PEMDAS order of operations is not violated.
- **Calculations Module**: This module is responsible for calculating equations. This module is designed to receive an operator and two operands and through a series of if statements facilitates operations such as addition, subtraction, or any custom-defined operator function.
- **Tree Module:** This core component is responsible for keeping/tracking the hierarchy of the PEMDAS order of operation for a given user input. As the parser module is extracting the equation for the tokenizer, it stores the operators and operands in a Binary Tree. This process represents the expression’s structure and allows for easy calculation of the equation using the Calculations module and tree traversal
- **Main**: This component manages user interaction, providing a user interface for data entry and result display. This allows for the seamless transfer of user input to the main logic of the system for processing.

Behaviorally, the system operates as follows:

1. The user inputs data through the CLI, which is captured and sent to the tokenizer which will return the tokens. Then tokens will be passed to the parse module.
2. The parse module analyzes all the tokens and enforces the PEMDAS order of operations. If the equation is valid, it will create a Binary Tree using the Tree module. If it is not it will return an appropriate error message.

3. After the Binary Tree is created, the tree will be evaluated recursively and each pair operator and operands will be calculated using the Calculations module will be done using.
4. The final result is passed back to the main function to be displayed to the user.

This architecture was chosen to ensure a modular, extensible, and efficient system. The use of trees allows for clear data handling, while the node-based design of operator classes facilitates future expansion. The modular approach ensures that each component functions independently, supporting easy maintenance and testing, making it well-suited for educational projects.

3. Architectural Goals and Constraints

The architecture of **Calc.co** is designed with the following goals and constraints in mind:

Goals:

- **Modular Design**: The system should be divided into clear modules (e.g., parsing, calculation, UI) for easy development and maintenance.
- **Extensibility**: The system should be easy to extend by adding new operators or modifying existing functionality.
- **Efficiency**: The architecture must support efficient real-time processing for a responsive user experience.
- **User-Friendly Interface**: The user interface should be simple and intuitive for easy interaction.

Constraints:

- **Technology**: The project must be implemented using C++
- **Time**: The project must be completed by the end of the semester, limiting the scope of the design.
- **Team Size**: The project is developed by a team of 5, which limits the complexity of the architecture.
- **Design**: The Tree class is implemented as nodes that are interconnected to form a hierarchical binary tree structure.
- **Performance**: The system must run efficiently on typical hardware available to most students.

4. Logical View

The logical view of **Calc.Co** illustrates the system's decomposition into subsystems and packages. Each package is designed to encapsulate specific responsibilities, which align with the key functionalities of the system: parsing input, performing calculations, and managing the user interface. This section outlines the overall package hierarchy and provides a detailed breakdown of the architecturally significant design modules or packages and their constituent classes.

4.1 Overview

- **Parsing and Tokenizer**: Processes the input by breaking it into manageable tokens, which are then used for parsing and further analysis.
- **Tree:** Organizes the data based on PEMDAS into a binary tree.
- **Calculation Engine**: Executes operations based on the operands and operators provided, ensuring that mathematical operations are correctly carried out.
- **User Interface (UI)**: Manages the interaction between the system and the user, displaying input fields and

4.2 Architecturally Significant Design Modules or Packages

Parsing, Tokenizer, Tree

Description: This package handles the processing of the input by breaking it into manageable tokens, which are then used for parsing and further analysis.

Key Classes:

Tokenizer
Description: The Parser class is responsible for interpreting the raw input provided by the user, separating operands from operators, and preparing the data for processing.

Responsibilities: Analyzes the input string and organizes the data into 4 categories NUMBER, OPERATOR, or PARENTHESES

Parser
Description: This process ensures that the user input is a valid expression and that the PEMDAS order of operations is not violated.

Responsibilities: Analyzing all the tokens and enforcing the PEMDAS order of operations or returning appropriate error messages.

Tree
Description: As the parser module is extracting the equation for the tokenizer, it stores the operators and operands in a Binary Tree. This process represents the expression's structure and allows for easy calculation of the equation using the Calculations module and tree traversal

Responsibilities: This core component is responsible for keeping/tracking the hierarchy of the PEMDAS order of operation for a given user input in the form of a binary tree.

Calculation Engine

Description: This package is responsible for executing the actual mathematical calculations using the operands and operators parsed from the tree.

Key Classes:

OperatorNode
Description: The OperatorNode class represents an operator and connects to operand nodes to perform calculations.

Responsibilities: Executes the operation by connecting operands and operators.

Calculations
Description: The CalculationHandler class coordinates the calculation process, interacting with the operator nodes and operands to generate the final result.
Responsibilities: Reads from the operand and operator queues and processes them for the final output.

User Interface (UI)

Description: This package is responsible for gathering user input and displaying the output results.

Key Component:

UI
Description: The UI component handles the input and output elements of the user interface, including fields for data entry and output display.

Responsibilities: Displays the user interface, gathers input and shows the result after processing.

Transfer
Description: The Transfer component is responsible for passing the user input from the UI to the core logic for tokenizing and parsing and then receiving the results.

Responsibilities: Passes input from the UI to the core system and outputs the results to the user.

5. Interface Description

User Interface (UI)

The UI consists of basic components for user interaction:

- **Input Fields**:
  - CLI input field that takes in a whole equation
- **Output Area**:
  - The area below the Input field that displays the result of the calculation.

Valid Inputs

- **Operands**: Numeric values (integers, decimals, or parentheses).
- **Operators**: +, -, *, /,%.
- **Input Format**: A string combining operands and operators, e.g., "3 + 5 * 2".

Resulting Outputs

Output: The result of the calculation will be shown as a number in the output area.

Example:

Input : "3 + 5 * 2"

Output: “13”

6. Quality

Extensibility
The system is built to easily add new features. For example, we can introduce new operators or change input methods without reworking the whole code, thanks to modular design.

Reliability
The architecture ensures reliable performance by isolating each component (like parsing and calculations). If something goes wrong in one part, it won't affect the entire system. We'll also handle errors, like invalid inputs, to keep the system stable.

Portability
The system will be portable since we're using a language that works across different platforms. It should run on various devices with little to no changes.

Security
Although the system doesn't require high-end security, we'll include basic input validation to avoid crashes or issues from bad inputs.