Team8 Co

Calc.co
Software Architecture Document

Version <1.0>

Revision History

Date	Version	Description	Author
10/Nov/2024	1.1	Completed the first version of the architecture design document	Kai

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:

- **Parse Module**: This module is responsible for analyzing user input and organizing it into two distinct queues: one for operands and one for operators. This process ensures that the input is structured correctly for subsequent operations.
- **do_calculations Module**: This core component executes operations by reading from the operand and operator queues. It utilizes operator classes, each structured as a node. Each node is designed to reference two operand nodes, facilitating operations such as addition, subtraction, or any custom-defined operator function.
- **Operator Classes (Nodes)**: Each operator is implemented as an independent class, forming a node structure. These classes handle specific operations and are connected to two operand nodes, supporting a flexible and modular operation process.
- **Sort Class**: This class is responsible for ordering operands in the queue, ensuring that operations are performed in the correct sequence as dictated by operation precedence.
- **UI Class**: This component manages user interaction, providing a user interface for data entry and result display. The transfer class within the UI module facilitates 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 UI, which is captured and passed to the parse module via the transfer class.
2. The parse module organizes the input into two queues: one for operands and another for operators.
3. The sort class processes the queues, arranging operands in the correct order for execution.
4. The do_calculations module reads from these queues and invokes the necessary operator classes, using their node-based structures to perform operations.
5. The final result is passed back to the UI for display to the user.

This architecture was chosen to ensure a modular, extensible, and efficient system. The use of queues 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**: Operator classes are implemented as nodes, and the system uses queues for operands and operators.
- **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 Queue Management**: Handles the parsing of input and organizes the data into two queues—one for operands and one for operators.
- **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 and Queue Management

Description: This package handles parsing the user input and organizing it into two distinct queues: one for operands and one for operators.

Key Classes:

Parser
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 two queues—one for operands and one for operators.

QueueManager
Description: The QueueManager class is responsible for sorting and managing the queues of operands and operators.
Responsibilities: Ensures that operands and operators are correctly organized based on precedence and ready for calculation.

Calculation Engine

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

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.

CalculationHandler
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 Classes:

UI
Description: The UI class 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 class is responsible for passing the user input from the UI to the core logic for parsing and calculation, and then receiving the results.
Responsibilities: Passes input from the UI to the backend 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**:
  - Operand input (for entering numbers).
  - Operator input (for entering mathematical operators).
  - A button to submit the input.
- **Output Area**:
  - Displays the result of the calculation.

Valid Inputs

- **Operands**: Numeric values (integers or decimals).
- **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"

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.