Arithmetic Expression Evaluator in C++

Software Architecture Document

Version 1.0

Revision History

| **Date** | **Version** | **Description** | **Author** |
| --- | --- | --- | --- |
| 11/10/2023 | 1.0 | The team members have provided comprehensive explanations and detailed descriptions, which pertains to the software architecture. | Gregory Markose, Taha Khalid, Siddh Bharucha, Saurav Renju, Alec Slavik, Divit Kannan. |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

1. Introduction 4

1.1 Purpose 4

1.2 Scope 4

1.3 Definitions, Acronyms, and Abbreviations 4

1.4 References 4

1.5 Overview 4

2. Architectural Representation 4

3. Architectural Goals and Constraints 4

4. Use-Case View 4

4.1 Use-Case Realizations 5

5. Logical View 5

5.1 Overview 5

5.2 Architecturally Significant Design Packages 5

6. Interface Description 5

7. Size and Performance 5

8. Size and Performance

8.1 Size Considerations 7

8.2 Performance Considerations 7

8.3 Target Performance Constraints 8

8.4 Optimization Strategies

9. Quality

Software Architecture Document

# Introduction

The Software Architecture Document provides a comprehensive overview of the architecture for the

project. The purpose is to capture and communicate the significant architectural decisions made to design

and construct the system.

## Purpose

The Software Architecture Document facilitates communication between the stakeholders by visualizing

the system’s architecture. It serves as a guide for system implementation by depicting how the software components and interfaces fit together. This document provides the basis for analysis and decision-making throughout the project’s lifecycle, ensuring technical consistency.

## Scope

This document encompasses the architecture of the project which is ‘Arithmetic Expression Evaluator in

C++’. It focuses on capturing and conveying the significant architectural decisions and component structure to provide systematic design guidance.

## Definitions, Acronyms, and Abbreviations

Can view the Project Glossary

## References

* Software Requirement Specification
* Iteration Plan
* Project Pan
* Design Document
* Glossary

## Overview

* Architectural Representation: This section elucidates the software architecture for the current system, detailing the necessary views and explaining the types of model elements each view contains.
* Architectural Goals and Constraints: This section outlines the software requirements and objectives with significant impact on the architecture.
* Use-Case View: This section lists and briefly explains use cases/scenarios that hold substantial architectural importance.
* Logical View: This section delves into the decomposition of the system into subsystems and packages.
* Interface Description: This section provides essential information for understanding the user-interaction with the system by focusing on major entity interfaces, including screen formats, valid inputs, and resulting outputs.
* Size and Performance: This section details the major dimensioning characteristics of the software that impacts the overall design. Hence, this is a crucial aspect of the architecture.
* Quality: This section ensures a holistic understanding of the system’s overall quality attributes by highlighting how the software architecture contributes to non-functional capabilities such as extensibility, reliability, and portability.

# Architectural Representation

* 1. Component Diagram
* Parser Component: Responsible for tokenizing and parsing arithmetic expressions.
* Evaluator Component: Evaluates the parsed expressions.
* User Interface Component: Manages user interactions and displays results.
  1. Module Diagram
* Lexer Module: Breaks down expressions into tokens.
* Parser Module: Converts tokens into a parse tree.
* Evaluator Module: Computes the result from the parse tree.
* UI Module: Handles user input and displays output.
  1. Class Diagram
* Expression (Abstract): Base class for various types of expressions (e.g., BinaryExpression, UnaryExpression).
* Parser: Parses expressions into an expression tree.
* Evaluator: Evaluates the expression tree.
* UserInterface: Manages communication with the user.
  1. Sequence Diagram
* User interacts with UI.
* UI triggers parsing by calling Parser.
* Parser generates an expression tree.
* Evaluator processes the tree.
* Result is sent back to UI for display.
  1. Deployment Diagram
* User's Device: Where the application is deployed.
* Arithmetic Evaluator Application: Executable running on the user's device.

# Architectural Goals and Constraints

**3.1 Software Requirements**

The architecture of the Arithmetic Expression Evaluator is guided by several overarching goals and requirements:

1. Extensibility:

The system architecture should support modular design principles, enabling easy integration of additional features and functionalities in the future. This allows for the seamless incorporation of enhancements such as new arithmetic operations or alternative input methods.

1. Reliability:

The architecture prioritizes reliability through comprehensive error handling mechanisms. The system must gracefully handle a variety of input scenarios, providing informative feedback to users in case of errors.

1. Portability:

To ensure widespread usability, the architecture emphasizes cross-platform compatibility. The implementation is designed to be independent of the underlying operating system.

**3.2 Architectural Goals**

The architectural goals of the Arithmetic Expression Evaluator a;ogms with other objectives to meet the project requirements effectively:

1. User-Friendly Interface:

The system aims to provide a command-line interface that could be engaged with by any person. The interface should be intuitive, with clear prompts and error messages, promoting ease of use.

1. Modular Design:

The architecture should focus on modularity to facilitate qualities such as future enhancements, ease of repair, and other modifications. Each module is designed to be separate, interchangeable, and modular as a component.

1. Security:

The architecture incorporates input validation mechanisms to prevent security vulnerabilities. Secure coding tactics should be followed with each module or design element. To protect the integrity of the system,

**3.3 Constraints**

Certain constraints influence the design and implementation of the Arithmetic Expression Evaluator:

1. Development Tools::

The project is constrained to use standard C++ as the primary programming language. This constraint ensures compatibility and simplifies the development process.

1. Minimalist User Interface:

The command-line interface design adheres to principles of simplicity and minimalism. This constraint is intentional to avoid unnecessary complexity and distractions for the end user.

1. Cross-Platform Compatibility:

The architecture is constrained to maintain compatibility across different operating systems. This is achieved by relying on portable C++ libraries and adhering to platform-independent coding practices.

# 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.]*

## 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.]*

# Logical View

The Logical View of the "Arithmetic Expression Evaluator in C++" design model unveils the system's structural organization. Subsystems and packages are strategically defined, each housing architecturally significant classes crucial for the calculator app's functionality. These classes carry specific responsibilities, such as parsing expressions, managing operators, and handling user input. Relationships between classes, essential operations, and attributes are meticulously outlined, contributing to the coherent design of the calculator app. This detailed perspective allows for a clear understanding of how classes collaborate within subsystems, ensuring the effective execution of arithmetic expression evaluation.

## Overview

The Overview section in the design model of the "Arithmetic Expression Evaluator in C++" project provides a holistic glimpse into the system's structural organization. It outlines the overall decomposition of the design model, emphasizing the package hierarchy and layers that form the backbone of the system. By presenting a high-level view of how packages are structured and layered, this subsection serves as a roadmap, guiding stakeholders through the system's architectural landscape. It lays the foundation for a deeper exploration into the subsequent sections, offering context and a preliminary understanding of the project's design structure.

## Architecturally Significant Design Modules or Packages

Within the detailed design model for the "Arithmetic Expression Evaluator in C++" project, this section meticulously dissects each architecturally significant package, providing an in-depth exploration of the project's structural backbone. Each package is introduced through a dedicated subsection, offering a comprehensive view that includes the package's name and a succinct yet informative description. To enhance understanding, visual aids are employed—diagrams depict the intricate relationships and dependencies among the significant classes and packages residing within each module.

Delving further, this section offers a granular view of each significant class within a package. For every class, the section furnishes essential information, including the class name, a concise yet detailed description, and, optionally, an exploration of major responsibilities, operations, and attributes. This comprehensive approach aims to empower stakeholders with a rich understanding of the project's architectural composition, ensuring clarity on both the macro-level package interactions and the micro-level nuances of individual classes.

PACKAGES

The "Arithmetic Expression Evaluator in C++" project, it's advisable to simplify the packages for ease of implementation, especially considering the coding experience of the team. Here are the revised, simplified packages:

* Tokenizer Module: Focuses on breaking down input expressions into individual tokens or elements.
* Expression Parser: Manages the representation of expressions using a straightforward data structure.
* Operator Handler: Deals with basic operator precedence and implements the logic for expression evaluation.
* Parenthesis Manager: Incorporates mechanisms to identify and evaluate expressions within parentheses.
* User Interface Module: Establishes a simple command-line interface for user input and result display.
* Error Handler: Implements basic error handling for scenarios like division by zero or invalid expressions.

# Interface Description

**Overview**

The Arithmetic Expression Evaluator interface is designed to be intuitive and user-friendly, ensuring that users can easily input their arithmetic expressions and receive accurate evaluations. The interface will primarily operate through a command-line interface (CLI), providing a clear and concise environment for expression input and result display.

**Input Interface**

* **Prompt Display:** The CLI will display a prompt such as “Enter an expression:” to indicate readiness for an input.
* **Expression Input:** Users can enter arithmetic expressions containing operators (+, -, \*, /, %, ^), numeric constants, and parentheses. For example, input can be 4 + 5 \* (6 - 3).
* **Input Validation:** The interface will validate the input to ensure it conforms to the acceptable format. Non-conforming inputs will trigger an error message, such as Invalid input format. Please enter a valid arithmetic expression.

**Future Extension**

**Graphical User Interface (GUI):** In future iterations, a GUI may be developed with text fields for input and buttons for operations, enhancing user experience and accessibility.

**Output Interface**

**Result Display**

* Calculation Result: After processing a valid input expression, the CLI displays the result in a clear format, like Result: 10.
* Error Messages: For invalid expressions or computational errors (like division by zero), the interface will display an informative error message. For instance, Error: Division by zero is not allowed.

Additional Features

* History: The interface will have the capability to display a history of recent expressions and their evaluations.
* Help Option: A help command can be included to guide users on how to use the interface effectively.

User-Interface Prototype Document

* A User-Interface Prototype Document (if available) should be referred to here for visual representations and further details about the interface design and layout.

**Technical Considerations**

* The interface will be designed to be platform-independent, ensuring compatibility across different operating systems.
* Special attention will be given to the responsiveness of the interface, ensuring minimal lag between input and output display.
* The interface design will adhere to the principles of simplicity and minimalism, avoiding unnecessary elements that could distract the user.

# Size and Performance

8.1 Size Considerations:

The Arithmetic Expression Evaluator project is designed to be lightweight and efficient in terms of size. The key size considerations include:

**Executable Size**:

* The goal is to maintain a compact executable size to facilitate quick loading and execution.

**Memory Footprint**:

* Strive for a minimal memory footprint to ensure efficient utilization of system resources.

8.2 Performance Considerations:

The performance of the Arithmetic Expression Evaluator is critical for providing users with a responsive and efficient experience. The primary performance considerations are:

**Expression Evaluation Speed**:

* The evaluator should execute arithmetic expressions swiftly to provide real-time results.

**Input Parsing**:

* Efficient parsing of user input expressions is essential for responsiveness.

**Error Handling Overhead**:

* Minimize the computational overhead associated with error handling to maintain performance under normal conditions.

8.3 Target Performance Constraints:

The target performance constraints for the Arithmetic Expression Evaluator are as follows:

**Expression Evaluation Time**:

* Aim for sub-second response times for typical arithmetic expressions.

**Memory Utilization**:

* Strive to keep memory usage within reasonable limits, ensuring compatibility with a wide range of computing environments.

**Platform Independence**:

* Maintain performance consistency across different operating systems to provide a uniform user experience.

8.4 Optimization Strategies:

To achieve optimal size and performance, the following strategies will be employed:

**Algorithmic Efficiency**:

* Implement algorithms for expression parsing and evaluation that are optimized for speed and minimal computational complexity.

**Resource Management**:

* Employ efficient resource management practices to minimize memory usage and enhance overall performance.

**Compiler Optimization**:

* Leverage compiler optimization flags and settings to enhance executable performance.

**Testing and Profiling**:

* Conduct rigorous testing and performance profiling to identify and address bottlenecks during development.

# Quality

**Overview**

The architecture of the Arithmetic Expression Evaluator is tailored to enhance system quality, focusing on extensibility, reliability, portability, and security, ensuring the system exceeds functional requirements.

**Extensibility**

* Modular Design: Components are separated for ease of modification and feature expansion.
* Interface Abstraction: Allows for future enhancements, like GUI integration, without major alterations to core logic.

**Reliability**

* Error Handling: Integrated throughout to manage exceptions and communicate effectively with users.
* Testing Framework: Comprehensive testing, including unit and integration tests, ensures reliability under various conditions.

**Portability**

* Cross-Platform Compatibility: Standard C++ usage ensures compatibility across different systems.
* Dependency Management: Portable libraries are used to maintain system portability.

**Security and Privacy**

* Input Validation: Prevents security vulnerabilities such as injection attacks.
* Data Handling: Adheres to secure coding principles to safeguard system integrity.

**Performance Optimization**

* Efficiency in Algorithms: Optimized for fast computations.
* Resource Management: Minimizes memory footprint and processor usage.

**Accessibility**

* User Interface Design: Focuses on ease of use, clear instructions, and straightforward navigation.

**Sustainability**

* Code Maintainability: Emphasizes clean, well-documented code for long-term sustainability.
* Environmental Considerations: Optimized performance reduces environmental impact.