Freaky Inc.

Freaky Calculator
Software Architecture Document

Version <1.0>

Revision History

Date	Version	Description	Author
<10/31/2024>	<1.0>	Initial Document	Abhiroop Goel
<11/04/2024>	<2.0>	Revised Document	Brady Boedy
<11/06/2024>	<3.0>	Finalized Revision	Sam Prestigiacomo

Table of Contents

Software Architecture Document

1. Introduction

The Software Architecture Document (SAD) provides a comprehensive overview of the architectural design decisions for the **Arithmetic Expression Evaluator (AEE)**, guiding stakeholders and developers toward a mutual understanding of project goals, architectural decisions, and development requirements. This document ensures consistency and alignment during development.

1.1 Purpose

The purpose of this SAD is to serve as a guiding reference, documenting architectural decisions and facilitating a shared understanding of the AEE's design principles and structure. It is intended to support development, testing, and future maintenance efforts, aligning all contributors with the project's requirements and objectives.

Scope

This SAD applies to the **Arithmetic Expression Evaluator (AEE)**, covering all aspects of its architecture, including the structural framework, goals, constraints, component interactions, and other considerations essential to its design and implementation.

1.2 Definitions, Acronyms, and Abbreviations

- SAD: Software Architecture Document
- AEE: Arithmetic Expression Evaluator
- IDE: Integrated Development Environment
- API: Application Programming Interface
- PEMDAS: Parentheses, Exponents, Multiplication, Division, Addition, Subtraction (order of operations)
- GoogleTest: A unit testing library used for C++ testing
- UML: Unified Modeling Language - A standard visual language for creating diagrams to represent software architecture and design
- Operator Overloading: The process of defining custom behavior operators when applied to user-defined types
- Template: A feature that allows functions and classes to operate with generic types

1.3 References

Software Engineering Body of Knowledge (SWEBOK), Version 3.0, IEEE Computer Society, 2014. This document provides standards for software engineering principles, including architectural design. It is available through IEEE Xplore Digital Library.

GoogleTest Documentation, Version 1.11, Google LLC, 2023. This is a comprehensive guide to the GoogleTest framework for unit testing in C++, essential for testing various components of the Arithmetic Expression Evaluator (AEE). It is available at: https://google.github.io/googletest/.

ISO/IEC 25010:2011 Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models, International Organization for Standardization, 2011. This outlines the quality standards applied in the SAD under the Quality section to ensure reliability, maintainability, and portability of the AEE. It can be found through through ISO's official website or local standards organizations.

Programming with C++: A Practical Introduction, 2nd Edition, John Smith, Pearson Education, 2021. Covers C++ best practices, including design patterns and modularization techniques referenced in the Expression Evaluation Package. It is found through Pearson's online catalog or academic libraries.

Unified Modeling Language (UML) User Guide, 2nd Edition, Grady Booch, James Rumbaugh, Ivar Jacobson, Addison-Wesley Professional, 2005. This was referenced for structuring UML diagrams in the SAD, especially for representing the architectural components and relationships in Section 2 and is available from major book retailers or university libraries.

1.4 Overview

The following document outlines the key elements required for understanding the architecture of the AEE. It provides views, goals, constraints, and a logical breakdown of the system's structure.

2. Architectural Representation

This section introduces the architectural model of the AEE, including major components and their interactions. The model is designed to allow for scalability, error handling, and flexibility in accommodating future change requests (e.g., supporting floating-point values).

3. Architectural Goals and Constraints

3.1 Goals

Expression Parsing Accuracy: Ensure accurate parsing and evaluation of arithmetic expressions, respecting operator precedence (PEMDAS) and grouping through parentheses.
Error Handling Robustness: Implement error handling for invalid expressions, division by zero, and syntax errors.
Modular Design: Structure components for easy extension and maintenance.

3.2 Constraints

Performance: Provide low-latency parsing and evaluation to deliver prompt results to users.
Command-Line Interface (CLI): The program will use a CLI for user interactions.
Integer Input Assumption: For the initial version, only integer inputs will be supported.

4. Use-Case View

Addition/Subtraction: Users can add or subtract two numbers.

Multiplication/Division: Users can multiply or divide two numbers. Division needs special handling for zero.

Exponentiation: Users can raise a number to a power.

Square Root: Users can find the square root of a number, considering inputs and non-negative values.

Modulus Operator: Users can find the remainder of two numbers being divided from one another.

PEMDAS: Calculator needs to function correctly according to the order of operations.

Overflow: Calculator needs to list large overflow numbers in a way that is easily readable to the users.

Invalid Input: Calculator needs to list an error message when the user inputs invalid operations.

4.1 Use-Case Realizations

PEMDAS: The calculator needs to know the correct order in which to operate an expression if multiple operators are used in the single expression. Performing operations in the correct order is crucial to finding a correct answer to the operation.

Overflow: The user multiplies two extremely large numbers together, resulting in an even larger number that cannot be fully listed on the display of the calculator. The calculator needs to display this number in scientific notation to allow the user to read the product of the two numbers.

Invalid Input: If the user is to input something into the calculator incorrectly, such as placing two operators right next to each other with no numbers to define the operation, the calculator needs to handle this error. It must first recognize the error then allow the user to correct the error or to skip over it and move on.

5. Logical View

This section provides a detailed look at the structure of the Arithmetic Expression Evaluator(AEE) by decomposing it into architecturally significant packages and classes. The design model is divided into the User Interface Package and the Expression Evaluation Package, each containing classes that fulfill specific responsibilities.

5.1 5.1 Overview

The AEE design is organized into two primary packages: the User Interface Package and the Expression Evaluation Package. These packages represent the different layers of the architecture, with the User interface Package handling input and output operations. The Expression Evaluation Package handles the core functionality of parsing and evaluating expressions. This layered structure allows for modularity, where user interaction logic is separated from the evaluation and error-handling logic.

5.2 Architecturally Significant Design Modules or Packages

The User Interface Package encompasses classes responsible for managing user interactions and providing feedback. The package ensures that expressions are correctly entered, validated, and results or error messages are displayed to the user.

6. User Interface Package

- **ExpressionController Class**
  Coordinates the actions of the user, serving as the central controller to manage expression entry, parsing, evaluation, and displaying results or errors.
- **ExpressionInput Class**
  Handles user input, validating expression format and ensuring input correctness before passing it to the parser.
- **ConsoleOutput Class**
  Displays messages, results, and notifications in the console. Manages the output format for both successful evaluations and error messages.

6.1.1.1 Expression Evaluation Package

- **Expression Class**
  Abstract class representing the base structure for an arithmetic expression, establishing the foundation for subclasses representing specific operators.
- **Operators (Subclasses)**:
  AdditionOperator, SubtractionOperator, MultiplicationOperator, DivisionOperator, ModuloOperator, ExponentiationOperator

  Represent specific operator types and encapsulate functionality to calculate results based on the operator's logic.

- **ExpressionParser Class**
  Parses user-entered expressions, applying operator precedence (PEMDAS) and organizing the expression in a structured format (e.g., expression tree) for evaluation.
- **ExpressionEvaluator Class**
  Evaluates the parsed expression structure by traversing or processing it a ccording to operator precedence and handling expressions with parentheses.
- **ErrorHandler Class**
  Detects and manages errors in user input, providing descriptive error messages to help users understand and correct issues.

Architecturally Significant Classes

- **Expression Class**

  Manages operands and operator relationships, serving as the base for arithmetic expressions. Establishes common evaluation logic used across operator subclasses to ensure consistent and accurate calculations.

- **ExpressionParser Class**
  Tokenizes the input, applies PEMDAS precedence rules, and organizes expressions into a structured format for evaluation. Handles parsing complexities, such as nested parentheses and operator ordering.

Important Relationships

- **ExpressionParser and ExpressionEvaluator Relationship**
  ExpressionParser produces a structured format (e.g., expression tree or stack) that is passed to ExpressionEvaluator for processing. This relationship is central to maintaining correct operator precedence and managing parentheses in evaluations.

- **Expression and Operators Relationship**
  The Expression class contains subclasses representing specific operators, which implement distinct arithmetic functionalities. This relationship allows each operator to follow a common interface while ensuring modularity for handling different operations.

6.2 Overview

5.1.2 User Interface Package

This package encompasses the user interface components responsible for facilitating user interactions and providing feedback on arithmetic expression evaluation.

Contained Classes: ExpressionController, ExpressionInput, ConsoleOutput

Responsibilities: The classes within this package handle user input, manage expression entry, and display evaluation results or error messages to the user. ExpressionController oversees the flow of interactions, while ExpressionInput validates the format of expressions. ConsoleOutput formats and displays system

messages and results.

5.1.3 Expression Evaluation Package

This package focuses on parsing and evaluating arithmetic expressions based on user input, adhering to operator precedence and handling grouping with parentheses.

Contained Classes: Expression, Operators, ExpressionParser, ExpressionEvaluator, ErrorHandler

Responsibilities: The classes within this package represent arithmetic operators, manage the parsing of expressions, and handle evaluation processes. The ExpressionParser tokenizes and organizes expressions into a structured form, while ExpressionEvaluator processes the parsed expressions. ErrorHandler provides robust error checking and generates user-friendly error messages for issues like syntax errors or division by zero.

6.3 Architecturally Significant Design Modules or Packages

5.2.1 User Interface Package Classes

ExpressionController:
Acts as the primary controller for user interactions, handling the overall flow of the application. It manages input from the user, initiates parsing and evaluation processes, and presents results or error messages in a structured format.

ExpressionInput:
Manages the input interface, where users enter arithmetic expressions. It validates input formats before passing them to the parser, ensuring they meet basic syntax requirements.

ConsoleOutput
Provides a console interface for displaying calculated results, system messages, and error logs. This class is responsible for formatting and outputting all system feedback to the user.

5.2.2 Expression Evaluation Package Classes/Subclasses

Expression
An abstract class representing a generic arithmetic expression. It defines the base structure and evaluation interface for expressions, with specific operators subclassed under this base class.

Operators
Subclasses of the **Expression** class, each representing a specific arithmetic operator. These include:

AdditionOperator: Handles addition operations.
SubtractionOperator: Handles subtraction operations.
MultiplicationOperator: Handles multiplication operations.
DivisionOperator: Manages division operations, ensuring checks for division by zero.
ModuloOperator: Handles modulus operations.
ExponentiationOperator: Manages exponentiation operations.

ExpressionParser
This class is responsible for parsing the input expression, breaking it into tokens, and creating a

structured representation that the **ExpressionEvaluator** can process. It applies operator precedence (PEMDAS) rules and handles grouping with parentheses.

ExpressionEvaluator
Evaluates parsed expressions, following the structure created by the **ExpressionParser**. It processes the expression tree or stack, applying operator precedence and handling recursive evaluations for expressions within parentheses.

ErrorHandler
Manages detection and handling of invalid expressions, such as syntax errors, mismatched parentheses, or division by zero. It generates clear error messages, allowing users to understand and correct their inputs.

7. Interface Description

6.1 Screen Formats

Command-Line Interface (CLI): The user interface operates through a command-line interface, allowing users to input arithmetic expressions and view the evaluated results.

Console Output: The CLI displays calculated results, error messages, and system status updates to inform the user of successful evaluations, parsing stages, or encountered errors.

6.2 Valid Inputs

Supported Operators: Users can input arithmetic expressions containing the predefined operators: addition (+), subtraction (-), multiplication (*), division (/), modulus (%), and exponentiation (**).

Operands: The evaluator accepts numeric constants as operands, including integer values. Parentheses can be used to group expressions and manage the evaluation order according to PEMDAS rules.

Command Syntax: Expressions must follow valid mathematical syntax (e.g., no consecutive operators without operands, perfectly balanced parentheses).

6.3 Valid Outputs

Evaluation Results: The CLI will output the calculated result of the expression, ensuring that it adheres to the order of operations.

Error Messages: For any invalid input, the evaluator will display an error message explaining the issue.

Errors may include descriptions for:

Syntax Errors: Issues such as unbalanced parentheses, unsupported symbols, or consecutive operators.
Runtime Errors: Situations such as division by zero or other invalid calculations.

8. Size and Performance

Size Characteristics

The **Arithmetic Expression Evaluator (AEE)** is designed as a lightweight command-line application. The size of the software is influenced primarily by:

Expression Complexity: The number of tokens (operands, operators, and parentheses) within an expression directly affects memory usage and processing time.
Expression Depth: Nested expressions with multiple levels of parentheses may increase the depth of the expression tree, requiring more memory to store intermediate nodes and more processing time for recursive evaluations.

Performance Constraints

The AEE targets efficient performance on standard hardware, aiming for near-instantaneous evaluation of typical expressions. Performance is influenced by:

Parsing Speed: The ExpressionParser is optimized to quickly tokenize and organize the input expression, applying operator precedence rules with minimal overhead.

Evaluation Speed: The ExpressionEvaluator is designed to process expressions based on a stack or expression tree, ensuring that the traversal respects operator precedence while handling parentheses efficiently.

Target Performance

Latency: The AEE should provide results within milliseconds for typical expressions, with complex, nested expressions (up to 10-15 operators and operands) targeted for sub-second evaluation times.

Error Handling: Error detection and reporting are expected to occur in real-time as the expression is parsed, ensuring that syntax and runtime errors are identified before evaluation begins.

9. Quality

8.1 Extensibility

The **Arithmetic Expression Evaluator (AEE)** is designed with a modular structure to support future expansion. The modular design allows for easy addition of new features, such as support for floating-point values, new operators, or complex functions. By following clean code standards and using encapsulated classes for each component , the architecture facilitates
straightforward updates and extensions without impacting existing functionality.

8.2 Reliability

The AEE will include robust error-handling mechanisms to ensure reliable performance. If an invalid expression is entered, the software should identify and describe the issue, whether it’s a syntax error, division by zero, or mismatched parentheses. Reliability testing will include:

- Verifying correct handling of various input scenarios.
- Testing for memory safety and addressing potential runtime errors.
- Ensuring system stability and performance across different expression complexities.

8.3 Portability

To maximize portability, the AEE will use standard C++ libraries exclusively, avoiding platform-dependent libraries or compiler-specific features. This approach ensures that the software can be easily adapted across different operating systems with minimal adjustments, making it versatile for deployment on both Windows and UNIX-based systems.

8.4 Safety & Security

While security needs are limited, a baseline access control mechanism will be implemented to prevent users from triggering unauthorized functions or manipulating the code's internal state. While data encryption is not essential, user input will be sanitized to prevent injection vulnerabilities or buffer overflow risks.