Group 0

Arithmetic Expression Evaluator in C++
Software Architecture Document

Version <1.0>

Revision History

Date	Version	Description	Author
19/09/2024	1.0	Initial Meeting	Evan Rogerson, Mason West, Nick Heyer, Rahul nesan, Ben Haney, Samantha Adorno
17/10/2024	1.0	Initial Meeting for part two	Evan Rogerson, Mason West, Nick Heyer, Rahul nesan, Ben Haney, Samantha Adorno, Zain Cheema
7/11/2021	1.0	Initial Meeting for part three	Evan Rogerson, Ben Haney, Mason West, Rahul nesan, Samantha Adorno

Table of Contents

Software Architecture Document

1. Introduction

The Introduction of this Software Architecture Document offers an overview of the system's architecture, capturing the key structural and behavioral components of the arithmetic calculator.

1.1 Purpose

It aims to capture and communicate the critical architectural decisions that inform the system's design, focusing on aspects such as expression parsing, operator precedence, error handling, and extensibility for future arithmetic functions.

1.2 Scope

The scope of this document is the Parser our team is tasked to create. This document references how we plan to build the parser and what it will look like architecturally. The structure and format of the parser is influenced by this document as well.

1.3 Definitions, Acronyms, and Abbreviations

Software Requirements Specification: (SRS) Use Case Modeling: The process of coming up with requirements and laying out the plan for the project. This entails interactions, dependencies, and other relationships between different actors. See the Project Glossary.

1.4 References

Documents referenced elsewhere in the Software Architecture Design may include the Project Plan and Software Requirements Specifications which can be found on the project GitHub page. This document may also make reference to the project rubric, which can be found on the EECS 348 Canvas page.

1.5 Overview

In order to organize our program, we will create a series of classes and functions for different parts of the program. This will include something for receiving input from the user, printing the output to screen, parsing the equation into something computer readable, and then a function to actually perform each operation.

2. Architectural Representation

The software architecture of the Arithmetic Expression Evaluator system is designed to provide a structured, modular, and maintainable approach for parsing and evaluating arithmetic expressions in C++. To effectively represent this architecture, we use multiple architectural views, each focusing on a distinct aspect of the system. These views help illustrate both the static structure and dynamic behavior of the system. Each view contains specific types of model elements, as follows:

1. Logical View:

Classes: These are the main building blocks of the system, defining entities like ExpressionParser, Evaluator, and various Operator classes (AdditionOperator, SubtractionOperator, etc.). Each class encapsulates specific responsibilities.

Attributes: These are variables defined within classes that store relevant data. For instance, ExpressionParser might have attributes for storing tokens, while Evaluator might hold the current expression being evaluated.

Methods: These are functions within classes that define behavior, such as parsing, evaluating, or performing an operation. Methods in ExpressionParser would parse input, while Evaluator would use Operator classes to execute calculations.

Relationships: These define how classes interact with each other, such as the dependency of Evaluator on specific Operator classes for performing operations. Relationships help illustrate how

components rely on one another to function.

2. Process View:

Interactions: These depict the exchanges between classes during runtime. For example, after ExpressionParser parses an expression, it passes it to Evaluator, which uses specific Operator classes to calculate results.

Messages: These are the method calls or data exchanges that occur between objects, such as Evaluator calling methods in Operator classes to perform addition, subtraction, etc.

Control flow: This includes elements like loops, branches, or conditionals to manage complex expressions with multiple operations or parentheses. Control flow ensures that expressions are evaluated in the correct order according to operator precedence and grouping rules.

3. Implementation View:

Modules: These represent individual files or code packages that contain classes and functions. For example, there may be a module for ExpressionParser, another for Evaluator, and separate modules for each Operator.

Dependencies: These are the relationships between modules, indicating how they rely on each other. For instance, Evaluator might depend on Operator modules to perform specific operations.

Interfaces: These are defined points of interaction between modules, ensuring that each part of the code communicates in a standardized way. Interfaces enable modularity and encapsulation, allowing different parts of the code to work together seamlessly.

3. Architectural Goals and Constraints

The architecture reflects key software requirements and objectives, such as security, portability, reuse, and potential integration with off-the-shelf products. Constraints include specific design strategies, development tools, team structure, schedule, and any legacy code considerations.

4. Use-Case View

We will not be using use case modeling, thus this section is not relevant.

4.1 Use-Case Realizations

We will not be using use case modeling, thus this section is not relevant.

5. Logical View

The parser design in C++ is organized into key subsystems for streamlined functionality: Lexical Analysis (Lexer), Syntax Analysis (Parser), Semantic Analysis, Error Handling, Symbol Table Management, and Utilities. Each subsystem has distinct responsibilities and interacts to support modularity, maintainability, and scalability.

5.1 Overview

The design model is organized into a package hierarchy with defined layers, outlining the overall structure and relationships between key components.

5.2 Architecturally Significant Design Modules or Packages

Each key package in the parser design includes its main classes and components, briefly describing their roles and interactions. Diagrams outline significant classes within each package, with notable classes

including descriptions of their responsibilities, core operations, and attributes.

6. Interface Description

This section provides an overview of the major interfaces for the arithmetic calculator, detailing input formats, valid inputs, outputs, and error handling.

Screen Formats: If a graphical interface is included, it may consist of a simple input field for users to enter arithmetic expressions, a set of buttons for each operation, and an output display for results.

Valid Inputs: Accepts standard arithmetic expressions, including integers, floating-point numbers, and basic operators (+, -, *, /), with support for parentheses to define precedence.

Input validation ensures that users are alerted when entering unsupported characters or incorrect syntax.

Resulting Outputs: Displays a numeric result, formatted according to input (integer or decimal).

Error messages are displayed for invalid inputs, such as division by zero or malformed expressions, providing guidance to correct the issue.

7. Size and Performance

The size of the parser shouldn’t be too large for any normal computer to handle. It will also not take up an unnecessary amount of space on the device. The performance of the parser will also be a reasonable amount of time. Delivering the computation in a desirable amount of time.

8. Quality

Our Object-Oriented approach ensures that our program is reliable and can handle any valid expressions it is given. It will be efficient and will not require an excessive amount of memory or processing power. Our C++ program will be portable, allowing it to operate efficiently on all common operating systems.