Team 19

EECS 348 Term Project
Software Architecture Document

Version <1.0>

Revision History

Date	Version	Description	Author
10/11/24	<1.0>	Created and filled out document	Team 19

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

The purpose of this Software Architecture Document is to serve as a key resource within the overall project documentation for the team. It bridges the gap between high-level requirements and detailed design within the arithmetic expression evaluator, providing the development team with a clear understanding of how the system is structured and how its components interact. This SAD will support project stakeholders, including developers, project managers, and testers, by ensuring that architectural decisions are aligned with the goals outlined in the Software Requirements Specification (SRS).

The intended audience for this document includes:

Developers: To understand the system's architecture and guide implementation.

Project Managers: To ensure that the project remains on track and aligns with initial architectural decisions.

Testers: To develop test plans that validate the architectural integrity and behavior of the system.

Stakeholders: To gain insight into the system's structure and how it fulfills the outlined requirements.

1.2 Scope

The Software Architecture Document (SAD) for the Arithmetic Expression Evaluator provides a detailed architectural overview essential for guiding the design and development phases. It applies specifically to the command-line-based expression evaluator, covering core subsystems like parsing, evaluation, error handling, and the user interface.

Impacted/Influenced Areas:

Design & Development: Ensures code alignment with architecture.
Integration: Facilitates compatibility with larger systems (e.g., compilers).
Testing/QA: Supports performance benchmarks and error handling validations.
Future Enhancements: Allows for scalability, such as float support.

1.3 Definitions, Acronyms, and Abbreviations

- SRS:
  - Software Requirements specification, a document that describes the requirements in detail
- PEMDAS:
  - The order of operations in arithmetic expressions - Parentheses, Exponents, Multiplication/Division, and Addition/Subtraction
- CLI:
  - Command-Line Interface, text-based interface where users input/receive information directly from a console
- C++:
  - A general-purpose programming language known for its efficiency and object-oriented programming
- Stack:
  - A Data structure that follows Last-In-First-Out, think a stack of plates
- Parsing:
  - The process of analyzing a string to identify components and structure
- Tokenization:
  - Breaking down a string into manageable parts (tokens) such as numbers, operators,

parentheses, etc
- Operator Precedence:
  - The rules that define the order in which different operations are performed in arithmetic
- Binary Tree:
  - A data structure in which each node has at most two children
- Modulo (%):
  - An operator that returns the remainder of a division operation
- Error Handling:
  - Mechanics in the program that detect and manage issues such as invalid syntax, unmatched parenthesis, or division by zero
- SAD
  - Software Architecture Design, a comprehensive document that provides an overview of the architecture of the system.

1.4 References

- C++ Standard Library Documentation
  - Available from https://en.cppreference.com/w/
- Project Introduction Document
  - 00-2024-EECS348-Term-Project.pdf
- Project Plan Document
  - 01-Project-Plan.docx
- Software Requirements Specialization
  - 02- Software-Requirements-Spec.docx

1.5 Overview

- Section 1: Introduction
  - This section introduces the SAD through the use of a purpose statement, a scope statement, a list of all the acronyms, definitions, and abbreviations, a references section, and finally this very overview section.
- Section 2: Architectural Representation
  - Describes the architectural views of the system and how the software architecture is represented. This section will explain the different model elements and the views necessary to capture the architecture effectively.
- Section 3: Architectural Goals and Constraints
  - Details the software requirements, goals, and constraints that have a significant impact on the system's architecture. This section includes considerations like safety, security, performance, portability, and development tools.
- Section 4: Use-Case View
  - Lists significant use cases or scenarios from the use-case model. It identifies scenarios that illustrate key aspects of the architecture or emphasize critical system functionality. This section highlights how the architecture supports different workflows within the system.
- Section 5: Logical View
  - Describes the major components of the design, including subsystem and package decomposition. This section presents architecturally significant classes and modules, along with their responsibilities and relationships, to give a clear picture of the system's structure.
- Section 6: Interface Description
  - A description of the major entity interfaces, including screen formats, valid inputs, and resulting outputs.
- Section 7: Size and Performance
  - This section discusses the major performance-related characteristics of the system, such as the dimensions of software components, processing time, memory usage, and

performance constraints.

- Section 8: Quality
    - Outlines the quality attributes the architecture aims to support, such as extensibility, maintainability, reliability, and portability. It also considers any specific quality concerns, like safety or security, and how the architecture addresses them.

2. Architectural Representation

- Logical view
    - Core components:
        - Tokenizer
        - Parser
        - Evaluator
    - Function:
        - Explains the separation of parsing, tokenization, and evaluation
- Process View
    - Runtime Behavior:
        - Seamless transitions from tokenization to parsing and evaluation
    - Function:
        - Creates efficiency with little interdependencies between stages
- Data Model View
    - Structures:
        - Token
        - Expression Tree
        - Operator Stack
    - Function:
        - Defines the handling of parentheses and operators (PEMDAS)

3. Architectural Goals and Constraints

- Goals
    - Performance:
        - Minimizing parsing and evaluation times
    - Accuracy:
        - Follows PEMDAS, and handling all edge cases (i.e. division by zero)
    - Modularity:
        - Ensuring additional operators are easy to implement
    - Usability:
        - Provides a clear method of use with feedback for valid statements and errors
- Constraints
    - C++ Implementation:
        - Making sure it is compatible with standard libraries
    - Team Organization:
        - Practicing an efficient streamline team effort
    - Memory:
        - Must be as memory efficient as possible
    - UI
        - Must operate in a standard CLI environments

4. Use-Case View

Use-cases include the user typing in a string of symbols such as (),,/,+,- or numbers 0-9. The input will then be parsed and processed by the app and output will be printed. In the case of an error, such as the user entering a symbol such as & or if the user types an invalid string, the program will display an error*

message.

4.1 Use-Case Realizations

Examples of use-cases could include:

- *The user types a valid input of (1+8)*3; the program would display 27.*
- *The user types an invalid input of (1-6/7; the program would display an error message such as "Parenthesis never closed."*
- *The user types an invalid input of &34-6%; the program would display an error message such as "Invalid symbol in input."*

5. Logical View

5.1 Overview

The program will consist of a single class, main, that holds the logic for the expression evaluator.

5.2 Architecturally Significant Design Modules or Packages

The main class will have the responsibility of taking in user input, parsing the input into tokens, then returning the correct mathematical answer to the user.

6. Interface Description

6.1 User Interface

Command-Line Interface (CLI): the evaluator will use a simple text-based interface that allows users to enter expressions. users will be prompted on the command line to enter an expression and will receive the feedback whether it be the output or an error in the command line.

- Input Format: Users can enter mathematical expressions that include numbers and operators such as (+,-,„/,%,**) and parentheses for grouping.
- Valid inputs: The system accepts integer based expressions ad float points
- Output: After evaluating the expression, the system will display the results on the CLI. if an error occurs the system will provide a detailed description of the error

6.2 **Software Interface:**

Integration with other modules: the evaluator is made to work as a part of a larger compile system.it can connect with other components such as the parse and the tokenizer modules, which break down the complicated expressions and make them easier to handle.

- **Function calls:**

  The evaluator will expose functions that accept tokens representing numbers, operators, and parentheses, allowing other parts of the compiler system to process expressions.
- **Error Handling and Reporting**: The evaluator will return specific error codes or messages for invalid expressions. Other modules in the compiler can use these error messages to handle or report issues as needed.

6.3 Input and Output Specifications

- **Input**: The user will enter a mathematical expression, which the evaluator parses and processes. Valid inputs include positive and negative integers and the operators and parentheses mentioned above.
- **Output**: The result of the arithmetic calculation or an error message if the input is invalid. Outputs are displayed directly in the command line.

7. Size and Performance

[A description of the major dimensioning characteristics of the software that impact the architecture, as well as the target performance constraints.]

8. Quality

- Extensibility: The software will have a modular design to make adding and editing operations quicker and easier.
- Reliability: The software will have built in error handling and input validation to ensure that nothing goes wrong in the program
- Performance: The software will use efficient algorithms to make parsing and other operations as fast as possible