Fancy Calculator Inc.

Arithmetic Expression Evaluator in C++
Software Architecture Document

Version <3.0>

Revision History

Date	Version	Description	Author
<10/24/2024>	<1.0>	Document Created	Luke Reicherter
<10/28/2024>	<2.0>	First Meeting Discussion	Ethan Le
<11/07/2024>	<3.0>	Second Meeting Discussion	Ethan Le

Table of Contents

Software Architecture Document

1. Introduction

This Software Architecture Document will detail the design of the Arithmetic Expression Evaluator in C++ and how it will all play together. All valuable information pertaining to this will be included in this document.

1.1 Purpose

The purpose of this document is to outline the design and organization of the Arithmetic Expression Evaluator in C++. This outline will be used during the development process to determine how the internals of the expression evaluator are designed.

1.2 Scope

This document presents a blueprint for developers, maintainers, and testers of this program to guide in its implementation. The main goal of this document is to ensure that the final product is easy to use, clear readability, efficient, and easily modifiable if necessary.

1.3 Definitions, Acronyms, and Abbreviations

SAD – Software Architecture Design
IDE – Integrated Development Environment

1.4 References

GitHub Repository: https://github.com/LukeReicherter/EECS-348-Group-Project

- Includes meeting logs and previous documents from the development process.

Visual Paradigm Online: https://online.visual-paradigm.com/

- Used to create diagrams within the Logical View section

1.5 Overview

The Software Architecture Design (SAD) for the Arithmetic Expression Evaluator is organized into distinct sections, each aimed at creating a compressive understanding of the software's architectural design. The document is structured as follows:

1. Introduction
2. Architectural Representation
3. Architectural Goals and Constraints
4. Logical View
5. Interface description
6. Quality

This structured approach ensures that the SAD document provides a thorough and detailed description of the Arithmetic Expression Evaluators' architecture, catering to the needs of stakeholders including developers, project managers and technical analysts.

2. Architectural Representation

The architecture of the Arithmetic Expression Evaluator in C++ will be separated into 4 main classes. These classes are as follows: Main, Tokenizer, Parser, and Evaluator. The names of these classes are subject to change. These classes will contain a specific section of the overall program process. Modular design will allow for better code readability and efficiency. Each class will have many subclasses that help achieve modular design. The descriptions of the classes are as follows:

- Main class
    - This class will handle the input from the user and send it to the tokenizer.

-
  - It will also print the result of the expression to the user's screen
- Tokenizer
  - This class will separate the expression into smaller parts that can easily be read by the parser
  - Ex: (4+7)/4 will become [(, 4, +, 7,), /, 4]
  - The tokenizer will also print an error message if an invalid expression is entered
  - The tokenized expression will then be sent to the parser
- Parser
  - This class will receive a tokenized expression and will sort through the input to find which parts need to be evaluated.
  - It will follow the PEMDAS order to ensure the expression is evaluated in the correct order
  - When the parser sends tokens to an evaluator, the parser will simplify the expression to the received output
  - Ex: [(, 4, +, 7,), /, 4] will become [(, 11,), /, 4]
  - Once the expression is completely evaluated, the solution is sent back to the main file to be printed
- Evaluator
  - The evaluator will be separated into many different operations
  - The Parser will send a simplified expression to the evaluator to determine the numerical answer
  - The operations must include addition, subtraction, multiplication, division, and exponentiation
  - An operation to simplify parenthesis will also be included
  - Ex: (11) will become 11
  - This class will also look for math errors such as dividing by zero
  - The output from any evaluation should always be a single number

3. Architectural Goals and Constraints

- Software Requirements
  - Functionality: The calculator must handle basic arithmetic functions and interpret rules such as PEMDAS, parenthesis, and numeric constants.
  - Operation Support: The calculator needs to be able to support each operand (+, -, *, /, %, and **). This includes addition, subtraction, multiplication, division, modulus, and exponentiation.
  - Security: The calculator needs to recognize division by zero, multiple pairs of parentheses, invalid characters and/or missing operands, as well as reliably handling operations.
- Special Constraints:
  - Off-the-shelf-product: Leverage any additional C++ libraries to speed up and simplify the process of creating the program. This will reduce the complexity of the program and ensure consistency by using common practices.
  - Portability: The program should not be platform specific. It should run smoothly whether the user is on MacOS, Windows, or Linux. Adhering to the programming languages' standards for platform compatibility will ensure reliability.
  - Distribution and Reuse: The program will use object-oriented principles so that it can be reused and/or potentially used inside of a larger program. Using comments and functions to create a clean program will help other programmers understand the code.
- Implementation Strategy:

- Tools: Standard IDEs such as Visual Studio Code will be used to create this program. This will streamline every step of the process (programming, debugging, and testing).
- Team Structure: Our team has been divided in a way that maximizes efficiency. Everyone will take a part in implementing the code, but the team is split up so that our strengths and weaknesses balance out by assigning specific roles to each person.
- Schedule: The full timeline has been set out with milestones and goals along the way. We've included requirements, documentation, implementation, debugging, and testing along the way to ensure that the final product will be delivered on time to present our best work.
- Legacy code: Any old software will be referenced, if necessary, but we must check compatibility beforehand. With new libraries and methods, it's likely legacy code will need to be rewritten in some way, but it can still provide insight.

4. Logical View

Our project design model is decomposed into several key packages and subsystems that aid in its functional components. The primary components are arranged into four main packages: main, tokenizer, parser, and evaluator. Each package contains specific classes and utilities used to aid the program's overall design.

These packages will use modular design by breaking down individual sections of the program into smaller classes that can run independently from the overall design. This will help achieve the goal of fast and efficient design and readability.

4.1 Overview

The main package of the program acts as the top layer, which will interface with the user. The tokenizer and parser will handle the preprocessing and structuring of the input expression. The evaluator will perform the arithmetic calculations of the expression. Each subsection of these layers is described in the section below. All naming schemes of variable/function/class types are subject to change throughout the design process, but the overall design should remain the same.

4.2 Architecturally Significant Design Modules or Packages

The main package handles the user's input, which is directed to the appropriate components. It will also display the output of the program to the user's screen.

- Main File
    - Handles user input and outputs to screen

The tokenizer package contains classes responsible for the breaking down of the input expression. This function will create distinct tokens (numbers, parenthesis, operators) that can be easily recognized by the parser.

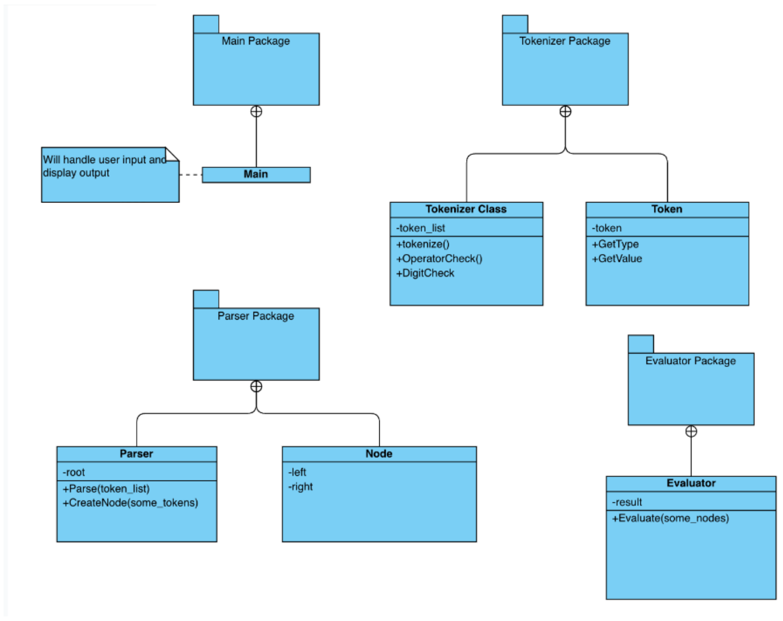
- Tokenizer Class
    - Tokenize() - Takes the input string from main and returns a list of tokens
        - Token Class – Each character from the input string will become a token.
            - GetType() - returns the type of the token
            - GetValue() - returns the value of the token if it represents a number
        - OperatorCheck() - Checks if the input character is an operator (May also include parenthesis)
        - DigitCheck() - Checks if the character is a digit

The parser package will analyze the tokens that have been created by the tokenizer. The tokens will be sorted into a tree-like structure in accordance with PEMDAS. This structure will be like the merge sort algorithm, where the initial list is broken down into individual parts and reassembled into the eventual answer.

- Parser Class
    - Parse(token_list) - Takes a token list and builds a tree structure according to PEMDAS order.
    - CreateNode(some_tokens) - Creates a Node that is a subset of the original token list.
        - Node Class – Contains a subset of tokens from the original list
            - Left – contains a subset of tokens of the original node
            - Right – contains a subset of tokens of the original node
    - Root – Top of the tree

The Evaluator package will be composed of multiple functions that take the input of two numbers and return the evaluated answer. The parser class will send the expression that must be evaluated to the evaluator as a node. This node must contain only a single operation (Ex: [1,+,2] NOT [1,/,2,+,3]). The evaluator will determine the necessary operation and send the surrounding numbers into the correct PEMDAS function.

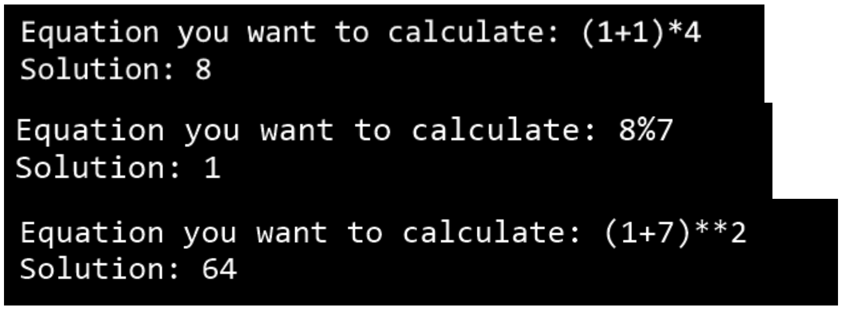
- Evaluator Class
    - Evaluate(Node) - Evaluates the equation of the current node
    - Result – stores the result of the equation



5. Interface Description

The interface will be formatted as a command line interface with two sections. The first section is where the user will input their desired equation, with the entire thing on one line. Below that, after the equation is given, a second section displaying the solution to their equation will be shown to the user.

Simple mockup of interface design with multiple different inputs (may change during development):



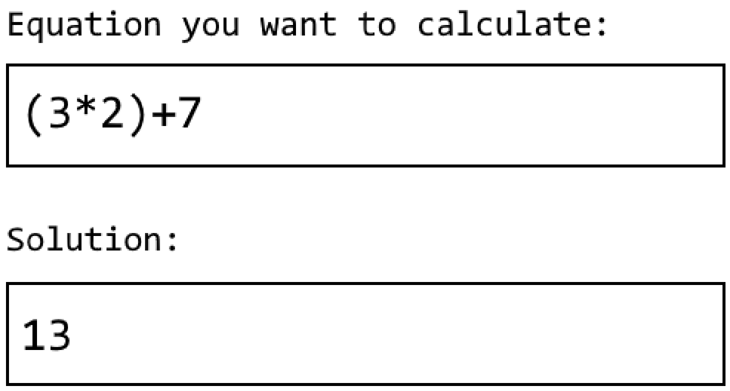
```
Equation you want to calculate: (1+1)*4
Solution: 8

Equation you want to calculate: 8%7
Solution: 1

Equation you want to calculate: (1+7)**2
Solution: 64
```

Optional alternative interface design:

- If time allows it, rather than the simple command line interface, a web interface with a GUI may be implemented. This would include the same two sections as the command line interface but would include separators for each section, rather than just a line break.
- Simple mockup of possible web interface design:



Equation you want to calculate:

(3*2)+7

Solution:

13

6. Quality

6.1 Extensibility

This program will use modular design to allow for easy modifications and additions. New operators can be added to the system by updating the tokenizer, parser, and evaluator. The tokenizer will need to recognize the new operator, the parser will need to assign the precedence of the operator, and the evaluator will need to calculate the answer.

6.2 Reliability

This program will retain reliability due to extensive error checking in the parser, tokenizer, and evaluator classes. The tokenizer will be responsible for checking invalid characters, the parser will check for incorrect syntax like unmatched parenthesis, and the evaluator will look for arithmetic errors such as dividing by zero. Comprehensive testing of the program will allow for all error possibilities to be found and dealt with.

6.3 Maintainability

Maintainability will be achieved through modular design and proper and complete documentation of the code. Each significant section of the program will need in-line commenting to facilitate possible future changes. Clear and consistent variable naming schemes will also contribute to the readability of the program.