NumberCrafters

Arithmetic Expression Evaluator
Software Architecture Document

Version <1.5>

Revision History

Date	Version	Description	Author
<12/11/23>	<1.1>	Added Quality and Interface Description	Blake Smith
<12/11/23>	<1.2>	Added 5.1-5.3	Landon Bever
<12/11/23>	<1.3>	Added Architectural Representation, Goals, and Constraints	Aidan Ingram
<12/11/23>	<1.4>	Added 1.1-1.2	Hale Coffman
<12/11/23>	<1.5>	Added 1.3-1.5	Kyle Spragg

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

This document identifies, introduces, and explains all the individual architectural components of our software. The views, model element, notations and documents will be identified and later described. This document will also showcase our Use-Case Model and describe, in detail, all the specific components of the model to display our software architecture. The goals and constraints are also identified and described. Finally, this document covers out logical view, interface and quality of our software.

1.2 Scope

Our Software Architecture Document applies to the Logical and Interface design of our software. This document affects each individual component of our software architecture by identifying and describing each component individually.

1.3 Definitions, Acronyms, and Abbreviations

A few items that should be more understood are the order of operations PEMDAS which highlights the order of basic operations: Parenthesis, exponents, multiplication, division, addition, and lastly subtraction. Other terms such as GUI (graphical user interface) is to acknowledge the aesthetics of this functionality, which may be very rudimentary since we are not using any GUI. Parsing means breaking down code into parts and describing the syntactical roles of said parts. Besides that, most of the items in this document are very straightforward and easily interpretable.

1.4 References

"EECS_348 Project Plan", Spragg, K., Coffman, H. Bever, L., Ingram, A., Smith, B.*, 9 October 2023,. Accessed 12 November 2023.*

"02-Software-Requirements-Spec" Spragg, K., Coffman, H. Bever, L., Ingram, A., Smith, B.*, 15 October 2023,. Accessed 12 November 2023.*

1.4 Overview

*[This subsection describes what the rest of the **Software Architecture Document** contains and explains how the **Software Architecture Document** is organized.]*

The Software Architecture Plan outlines the structure, design, and goals of the calculator software being developed in C++. The document is organized into several sections covering architectural representation, model elements, notations, architectural goals, constraints, team structure, schedule, and more.

- Architectural Representation:
  - Module, component and connect, and deployment views
- Architectural Goals and Constraints
  - Emphasis on safety, security, privacy, integration, portability, distribution, and reuse.
  - Constraints related to GUI absence, data collection, C++ compatibility, GitHub hosting, and team schedules.
- Logical View
  - Subsystem decomposition into Parser and Evaluator
  - Package decomposition with significant classes in the Evaluator package

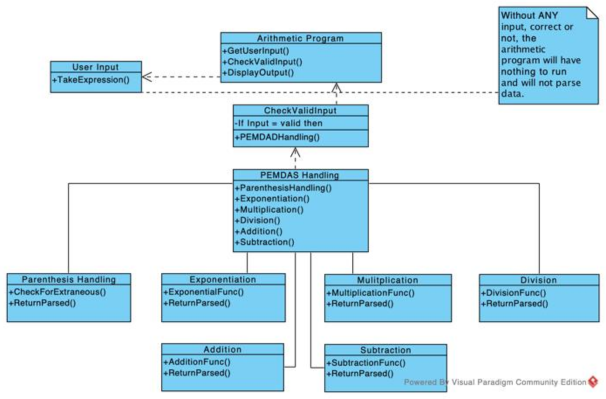
- Package Hierarchy
  - Main packages: ExpressionEvaluatorApp, Parser, Evaluator, Utilities, and Main
- Design Modules or Packages
  - Focus on the Evaluator package
- Interface Description
  - User-friendly interface for arithmetic expression input and result display
  - Seamless interaction with parser and constants
- Quality
  - Emphasis on extensibility, reliability, and portability
  - Error handling for safety and security
  - Modular design for future changes
  - Use of C++ for cross-environment portability

2. Architectural Representation

- **Views**
  - Module View - The software we are developing is intended to be an arithmetic calculator, implemented in C++. From the Module View perspective, the individual components relating to the arithmetic calculator can be broken down into different areas of functionality, and hierarchically categorized. Beginning with the top of the hierarchy is a front-facing, fully implemented calculator, from which it is broken down into components concerning individual functions such as: Multiplication, Division, Addition, Subtraction, Exponentiation, and Parenthesis Handling. Following these individual components in the hierarchy are the components that make up the actual code including the PEMDAS format, and individual design decisions. From the functional arithmetic calculator down, the hierarchy can be seen to end at the beginning of the implementation of the functional code.
  - Component and Connector View - This view focuses on the runtime behavior of our implemented code, the components, and the connectors that enable them to interact with one another. Our components in this software can be seen to be the individual methods that concern PEMDAS functionality (multiplication, division, etc.), and their connectors are the classes we will implement in the parsing of the user's input, to determine which part of the methods functionality should be utilized. Our runtime behavior should be normal, and memory and time complexity should likely never exceed some O(n) complexity.
  - Deployment View - This view concerns the physical deployment of our software system, including hardware use cases, and how different software components are distributed across different nodes. Concerning this project specifically, our physical implementation is very minimum as project standards only require us to have a single interface in the terminal for the final product, and thus there are no software components distributed across different nodes of our creation.
- **Model Elements**

- Components – The components of our software in terms of system architecture are primarily, as defined above in the views section, our functionality through PEMDAS and other methods that allow the user to receive an output that they find to be correct, no matter what arithmetic expression is used as input. Specifically defining our components shows that we have classes of Multiplication, Division, Addition, Subtraction, Exponentiation, and Parenthesis Handling. These components make up the bulk of our software.
- Connectors - Our connectors largely concern data passage and parsing, as there is a central connection between all classes that allows the program to distinguish between input that contains different classes and functionality. This is implemented in our code as a selection of calls in our main.
- Interfaces - Our interfaces, or how our components interact with each other, is simple as we are running one C++ program with built in interaction in the files. They are all formatted in .cpp file data, and there is no parsing done through JSON files or other data types.
- Nodes - As mentioned above, we have very little physical or server implementation work to do with this project, and thus we don't have many nodes to discuss within our deployment view.

- **Notations**
  - Diagrams - To represent our class hierarchy we have designed one central UML Class Diagram for our example, as shown below.



  - Documentation – This section will serve to elaborate further on our design decisions in our UML Class Diagram above. As is seen, our central point of the class diagram is our arithmetic program module. From here we call User Input to ensure we have possible input, and from there confirm that the input is valid. If we have an input and it's valid, we should then utilize our PEMDAS Handling to return a parsed function and display the output. This diagram, while rudimentary, shows full functionality of our modules in action.

3. Architectural Goals and Constraints

- **Safety and Security**
  - Our goal: Provide feasible and useful security to the user that allows them to avoid entering personal data and provide our code in a way that does not allow for direct manipulation.

    - Potential constraints: We are not implementing this code with a web app or GUI, so we need to consider if our user might want to interact with our source code, or source .cpp file, and how we can avoid that.
- **Privacy**
    - Our goal: User privacy should be entirely protected with no data collection or attempts at overarching consumer control.
    - Potential constraints: Our project guidelines do not require user recognition or cookie collection, so we would need to avoid logging who uses our source code at any given time.
- **Integration**
    - Our goal: Allow our software to be runnable on any machine with C++ and GCC compiler installed, while avoiding massive errors in production.
    - Potential constraints: Project guidelines mandate that our code be implemented in C++ and runnable from the terminal; easily met requirements.
- **Portability**
    - Our goal: Our code can be run on any OS or machine running C++ and GCC compiler, regardless of location or user.
    - Potential constraints: C++ 98 and versions before differentiating from C++ 11-23, and thus we will need to consider the potential for a user to be running an older version of C++ with our program.
- **Distribution**
    - Our goal: Allow our code to be hosted on Github.com and be hosted in a public fashion, allowing pull requests from any location.
    - Potential constraints: Project guidelines indicate our code MUST be in a GitHub repository, and thus cannot be hosted on Docker or related web-app sites.
- **Reuse**
    - Our goal: Write sustainable code that can be utilized in other projects, by us, through a header file that provides the ability to parse arithmetic functions, without redundancies.
    - Potential constraints: Project guidelines do not dictate how our code should be written in style or functionality, and as our group are C++ beginners, we must ensure that redundancies are kept to a minimum with constant code reviews.
- **Team Structure**
    - Our goal: Create a functional group with proper leadership and delegation, that meets regularly to discuss project requirements.
    - Potential constraints: People live in different areas, and everyone is very busy trying to be a successful STEM student.
- **Schedule**
    - Our goal: Release a new iteration of our code once every two weeks, as is dictated in previous project documents.
    - Potential constraints: Timing with other classes programming our personal business could delay launches, as well as individual issues with the code.

- **Legacy Code**
    - This section is not applicable to us, as we discuss how we will document our code that is entirely new in our Reuse section, to avoid legacy code that is dated and inadequate for future use.

4. Logical View

1. **Subsystem Decomposition:**
    - **Parser Subsystem:**
        - Responsible for parsing input arithmetic expressions.
    - **Evaluator Subsystem:**
        - Handles the evaluation of parsed expressions.
2. **Package Decomposition:**
    - **Parser Package:**
        - Contains classes responsible for lexical analysis and syntax parsing.
            - Syntax Analyzer
            - Expression Tree Builder
    - **Evaluator Package:**
        - Contains classes responsible for expression evaluation.
            - Operand Handler
            - Operator Handler
            - Expression Evaluator
3. **Architecturally Significant Classes:**
    - **Expression Evaluator Class:**
        - **Responsibilities:**
            - Evaluating arithmetic expressions.
        - **Important Relationships:**
            - Depends on Operand and Operator Handler classes.
        - **Key Attributes:**
            - Expression tree, current result.
        - **Key Operations:**
            - **evaluateExpression()**
4. **Class Utilities:**
    - **MathUtility Class:**
        - **Responsibilities:**
            - Provides mathematical operations.

- **Important Relationships:**
  - Used by various classes in the Evaluator subsystem.
- **Key Operations:**
  - **add()**, **subtract()**, **multiply()**, **divide(), exponent(), modulo()**

5. **Relationships, Operations, and Attributes:**
   - **Expression Evaluator Class:**
     - **Important Relationships:**
       - Depends on Operand and Operator Handler classes.
     - **Key Operations:**
       - **evaluateExpression()**
     - **Key Attributes:**
       - Expression tree, current result.

4.1 Overview

1. **Package Hierarchy:**
   - **ExpressionEvaluatorApp:**
     - The top-level package representing the entire application.
     - Subpackages:
       - **Parser**: Contains classes related to the parsing subsystem.
       - **Evaluator**: Contains classes related to the evaluation subsystem.
       - **Utilities**: Contains shared utility classes.
       - **Main**: Contains the main application entry point.
   - **Parser:**
     - Responsible for lexical analysis and syntax parsing.
     - Classes:
       - **SyntaxAnalyzer**: Analyzes the syntax of the input.
       - **ExpressionTreeBuilder**: Builds expression trees.
   - **Evaluator:**
     - Responsible for evaluating arithmetic expressions.
     - Classes:
       - **OperandHandler**: Handles operands during evaluation.
       - **OperatorHandler**: Handles operators during evaluation.
       - **ExpressionEvaluator**: Orchestrates the overall expression evaluation.
   - **Utilities:**

- Shared utility classes used across subsystems.
- Classes:
  - **MathUtility**: Provides mathematical operations.
- **Main:**
  - Contains the main application entry point.
  - Class:
    - **MainApplication**: Initiates the expression evaluation process.

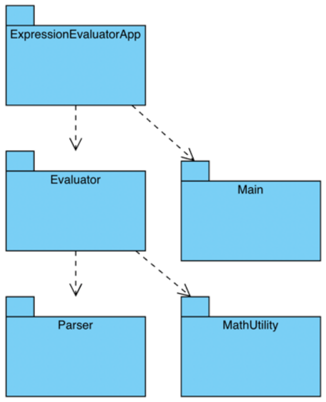
2. **Layers:**
   - **Presentation Layer:**
     - Interface with the user.
     - Contains the MainApplication class from the Main package.
   - **Application Layer:**
     - Implements the application logic.
     - Contains the ExpressionEvaluator class from the Evaluator package.
   - **Domain Layer:**
     - Represents the core domain logic.
     - Contains classes such as SyntaxAnalyzer, OperandHandler, and OperatorHandler.
   - **Infrastructure Layer:**
     - Provides supporting utilities.
     - Contains the MathUtility class from the Utilities package.

4.2 Architecturally Significant Design Modules or Packages

1. Evaluator Package

- Overview:

  The Evaluator package is responsible for evaluating arithmetic expressions.
- Diagram:



- OperandHandler Class
- **Description:**
    - Handles operands during evaluation.
- **Responsibilities:**
    - Managing operands during expression evaluation.
- **Operations:**
    - **handleOperand()**: Processes individual operands.
- **Attributes:**
    - **currentOperand**: Holds the current operand.

OperatorHandler Class

- **Description:**
    - Handles operators during evaluation.
- **Responsibilities:**
    - Managing operators during expression evaluation.
- **Operations:**
    - **handleOperator()**: Processes individual operators.
- **Attributes:**
    - **currentOperator**: Holds the current operator.

ExpressionEvaluator Class

- **Description:**
    - Orchestrates the overall expression evaluation.
- **Responsibilities:**
    - Coordinating the evaluation of arithmetic expressions.
- **Operations:**

  - **evaluateExpression()**: Initiates the expression evaluation process.
- **Attributes:**
  - **expressionTree**: Stores the expression tree.
  - **result**: Holds the current result of the expression evaluation.

5. Interface Description

[A description of the major entity interfaces, including screen formats, valid inputs, and resulting outputs. If a User-Interface Prototype Document is available, refer to it in this section]

The user interface of our arithmetic expression evaluator is designed to be user-friendly as well as straightforward. When the application is opened, you'll see an input screen where you can easily type in your arithmetic expressions. Operators such as +, -, *, /, %, and & will be used, as well as parentheses for grouping. Once you enter your expression, a result display screen will show you the calculated result or any error messages if the expression is invalid. If any issues are encountered, the application will provide clear guidance on correction of your error. Our system interface will work seamlessly with other components such as the expression parser and numeric constants to ensure accurate and efficient expression evaluation.

6. Quality

[A description of how the software architecture contributes to all capabilities (other than functionality) of the system: extensibility, reliability, portability, and so on. If these characteristics have special significance, such as safety, security or privacy implications, they must be clearly delineated.]

The architecture ensures extensibility through a modular approach, allowing for straightforward integration of potential change requests, such as the future inclusion of the "**" operator for exponentiation. Reliability is a central focus, with error-handling mechanisms implemented to manage scenarios like division by zero or invalid expressions. Emphasis on reliability ensures accurate and dependable results, improving the overall quality. The architecture is also geared towards portability, using C++ to create a solution that can be run across different environments. The modular design ensures that updates, particularly those related to security measures for handling future changes, are clearly delineated. Error-handling mechanisms not only contribute to reliability, but also address safety concerns such as preventing division by zero.