Whatever Co.

Whatever Calculator
Software Architecture Document

Version 1.0

Revision History

Date	Version	Description	Author
07/11/24	1.0	First version of the Software Architecture Design	All Team Members

Table of Contents

Software Architecture Document

1. Introduction

The *Software Architecture Document* (**SAD**) outlines the architectural design and structure of the arithmetic parser, *Whatever Calculator*. This document will include the architecture principles to ensure efficiency and accuracy in its calculations, and serve as a guide for the development team, stakeholders, and testers by providing a solid understanding of the system's expected performance and capabilities.

1.1 Purpose

The purpose of the **SAD** is to provide a clear blueprint for developing the *Whatever Calculator*, detailing design decisions, and structural guidelines. The **SAD** will aid developers, project managers, and stakeholders, ensuring the calculator yields consistent outputs, is easy to maintain and use, and is adaptable for future updates. This document will include what software architecture the arithmetic evaluator uses, the use-case and logical views, as well as goals and constraints of the project related to the architecture, subsystems, and the interface.

1.2 Scope

The **SAD** will cover the architectural specifications for the *Whatever Calculator* such as the core functionality, modular components, error handling, user interface, and performance. This document will support the lifecycle of the calculator, ultimately helping stakeholders understand the design and provide guidance for development and maintenance.

1.3 Definitions, Acronyms, and Abbreviations

The project**:** "Whatever calculator", described by the Software Development Plan (**SDP)**. The repository for all code and references at https://github.com/i662m589/EECS348TermProject

SDP: Software Development Plan. *See Section 1.4 References - SDP*
SRS: Software Requirements Specification. *See Section 1.4 References - SRS*

SAD: Software Architecture Document. *See Section 1.4 References - SAD*

1.4 References

SDP : https://github.com/i662m589/EECS348TermProject

SRS:https://github.com/i662m589/EECS348TermProject

SAS: https://github.com/i662m589/EECS348TermProject

1.5 Overview

The document is organized into sections, with each section covering different topics concerning the Architectural design of this project. This document covers sections over Architectural Representation, Architectural Goals and Constraints, Logical View, Interface Description, and Quality. Each section has its own purpose in this document giving a better understanding of the project's overall architecture and design.

2. Architectural Representation

This software uses a pipe-and-filter style architecture, which can be represented as a collection of subsystems, arranged in a sequence (such as the diagram in section 4.1). This software will be examined from a logical view and from a development view. The purpose of the logical view is to enumerate the steps that need to take place for the software to function; on the other hand, the development view prevents the development of the code from becoming unwieldy due to poor file organization. The logical view consists of the various subsystems used in the program, and is covered more in-depth on section 4. The development view will consist of the directories that are in the Github repository, and will be briefly covered here.

In the GitHub repository, there will be one folder for each subsystem mentioned in the logical view, as well as a folder for the project artifacts (e.g. this document). Each subsystem's folder contains any code necessary for that subsystem to function, and may be further organized as the need arises. In addition, the file with the main function and the makefile will be placed at the root directory.

3. Architectural Goals and Constraints

Architecture Goals:

- Accuracy: Ensure calculations are performed with high precision, especially floating-point values.
- User-friendly interface: Provide an easy to use, simple user interface. Also ensure efficiency for outputs.
- Portability: Allow the calculator to run through multiple platforms (laptop, desktop, etc.).
- Reusability: Design core functionalities to be modular, enabling reusability and future modifications.

Architecture Constraints:

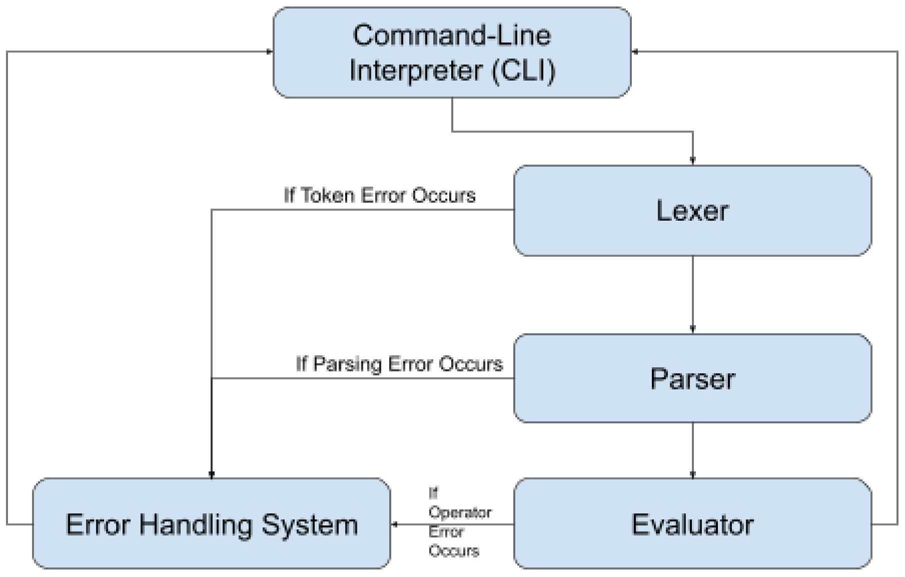
- Design Strategy: Use object-oriented principles to design codebase.
- Development tools: Keep in mind that code may need adjustments from running on different platforms. Also, C++ includes manual memory management.
- Legacy Code: Making sure future code can easily be integrated with the first version.

4. Logical View

4.1 Overview

The package hierarchy of this project includes five major components and layers. These five components are the Command-Line Interpreter (CLI), the Expression Lexer, the Expression Parser, the Expression Evaluator, and the Error Handling system.

Here is an image showcasing the package hierarchy of the system:



4.2 Architecturally Significant Design Modules or Packages

Command-Line Interpreter (CLI):

The CLI is the component that will interact directly with the user to get their input. In this component, the

CLI will print a message to the user stating that they have two options to interact with the program:

1) Enter an arithmetic expression so that the program can evaluate and print the result of the expression back to the user
2) Enter 'exit' so to quit the program

If the program receives an expression or input that isn't 'exit', it will take that input and send that input value through a pipe to the Lexer, where the Lexer will then interact with that expression. If the user enters exit, the program will subsequently end its run-time.

Expression Lexer:

The Lexer is the component that will read through the input and break down each individual character in the input into its own individual token. For example, the input '1+2' will be broken up into three tokens, which are '1','+' and '2'. This component will also be able to read if a token is an invalid arithmetic operator or expression such as '!' or 'a'. If the value has an invalid token, the Lexer will pipe this information to the Error Handling System. If all the expression's tokens are valid, the Lexer will pipe the expression into the Parser.

Expression Parser:

The Parser is the component that will parse through the expression and find the order in which to do the operations given in the expression. This will be done by following PEMDAS, where the parser will parse expressions in parentheses first, search for multiplication/division from left to right, and then search for addition and subtraction from left to right. The parser will put the order the expressions need to be parsed into a binary tree, which will be then passed to the evaluator. If there is an invalid format for the parser, such as a parenthesis without a pair, then the parser will hand this information to the error handling system.

Expression Evaluator:

The Evaluator is the component that will evaluate each of the expressions from the tree that is handed from the parser. This evaluator will go through each of the operations in the tree until all of the operations in the tree have been performed, yielding a final result. If there are any invalid operations that are performed, such as dividing a numeric value by zero, the evaluator will hand this information to the error handling system. Otherwise, the evaluator will get a result from the operations performed on the tree and display it to the user. The evaluator then hands control back to the CLI after it displays the result.

Error Handling System (EHS):

The Error Handling System (EHS) is the component that detects whether the expression that was inputted by the user was valid or not. If the expression is invalid, the EHS will cancel the current operation being performed and display the error to the user using the information that can be given from each of the components. If the Lexer tells the EHS that there is an invalid token, the EHS will display to the user that there is an invalid token in the expression. If the Parser tells the EHS that the expression that was entered is in an invalid format, the EHS will display that the expression is in an invalid format. If an invalid operation was performed, the EHS will display that there was an invalid operation that was performed. After displaying this information, the EHS will hand control back to the CLI.

<u>A diagram of each components interactions with each other is shown in Section 4.1</u>

5. Interface Description

The interface for the Whatever Calculator is a Command-Line Interface(CLI). A CLI is a straightforward format where user input and system output are displayed line-by-line, making interactions readable and user-friendly

. When the calculator is launched the CLI will display a message that will briefly describe the Whatever

Calculator. This message will include instructions on how to use the calculator as well as information about how to exit.

screen example:

```
Welcome to the Whatever Calculator!
Enter arithmetic expressions to evaluate them. Type 'exit' to quit.

Enter expression: 3 + 4 * 1
Result: 7

Enter expression: exit
Goodbye!
```

The user can input any expression they want but for it to be evaluated it has to be a valid expression within the limits of the calculator. Valid inputs are expressions that:

- Follow basic arithmetic rules (e.g., no division by 0)
- Follow formatting conventions (e.g., balanced parentheses)
- User operations that the calculator supports(+, -, *, /, %, **)

If the user follows these rules and input a valid expression they can expect the result of their expression to be displayed right below as shown in the example. If the expression is invalid they can expect to see an error message explaining why the expression was invalid.

6. Quality

- **Extensibility**
  - As future requirements may introduce themselves the chosen modular architecture allows for adding features with minimal modification to already existing modules of the program. For example the support of a new operation would only require creating a new module for said operator and modifying the parser to recognize the new operator.
- **Reliability**
  - Having a modular approach to creating the Whatever Calculator enables to handle each component independently. This separation of concerns makes the code easier to understand as well as simplifies debugging. By isolating individual functionalities the program becomes more reliable because each module can be tested in isolation, ensuring potential issues are identified early so they do not pograte through the entire program.
- **Portability**
  - A modular approach guarantees portability at the software and system levels. By breaking the calculator into separate modules, each component could be reused in other programs in the future. At the system level, as long as we adhere to C++ standards, the Whatever Calculator can be made portable across different systems..
- **Performance**
  - By separating each component, it becomes easier to design more efficient algorithms for parsing and evaluating. Breaking the program into unique modules encourages code reusability and, as mentioned earlier, enhances maintainability.