University of Kansas EECS 348

Arithmetic Expression Evaluator in C++
Software Architecture Document

Version 2.0

Revision History

Date	Version	Description	Author
10/28/24	1.0	Filled out 1.1	Tyler
10/28/24	1.1	Filled out section 2	Eliza
10/29/24	1.2	Filled out 1, filled out title page/header	Lillian
10/29/24	1.3	Filled out 1.5	Max
10/29/24	1.4	Filled out section 3 and revised table of contents	Eliza
10/31/24	1.5	Filled out section 5 and outlined section 5.2	Eliza
10/31/24	1.6	Filled out section 4.2 and added to 5	Tyler
11/1/24	1.7	Filled out section 4.1	Lillian
11/1/24	1.8	Filled out section 4.2 and added to 7	Max
11/4/24	1.9	Filled out 5.2 - Modulus	Tyler
11/5/24	1.10	Filled out section 5.2-Input Processor and section 6	Lillian
11/5/24	1.11	Filled out sections 1.3, 1.4, 5	Lillian, Eliza, Tyler, Daniel
11/7/24	1.12	Filled out section 7	Max
11/8/24	2.0	Final Edits	Lillian, Eliza

Table of Contents

Software Architecture Document

1. Introduction

The Software Architecture Document (SAD) provides a comprehensive architectural overview of the system in development (the Arithmetic Expression Evaluator). The SAD contains several different architectural views to fully depict each possible use case of the system. The SAD will contain the relevant and significant architectural decisions made concerning the system. Utilization of this document during the construction process will ensure that the finished system fulfills all requirements outlined in the relevant University of Kansas Electrical Engineering and Computer Science Software Engineering I (KU EECS 348) Canvas assignments.

1.1 Purpose

The purpose of this software architecture document is to provide a structural framework that will guide the development of the Arithmetic Expression Evaluator (AEE). Additionally, this document will provide clear instructions and guidelines that will be used during the implementation of the AEE. The AEE development team will often refer to this document during the project's implementation phase to ensure that the project remains true to its development goals. This includes following the structural requirements and restrictions that are set in this document.

Additionally, this document provides an overview of how each part of the complete system will work together. This ensures that the system will evenly distribute responsibilities among smaller subsystems. Designing the AEE architecture in this way provides a modular structure that can be easily changed to fit development needs.

1.2 Scope

The information in this document will provide a framework for how the AEE will be organized in terms of modules (functions and/or classes) to be made for the AEE to function. Modules and their functions in this document will be defined in detail and may provide insight into how they should be developed and used in the program. All modules defined in this document will be essential to the functionality of the AEE, and will directly address requirements outlined in previous documents.

1.3 Definitions, Acronyms, and Abbreviations

AEE- Arithmetic Expression Evaluator in C++; the name of the project outlined in the SAD document

EECS- the department of Electrical Engineering and Computer Science at the University of Kansas

EECS 348- class number for Software Engineering I at the University of Kansas

int- integer

KU- University of Kansas

SAD- Software Architecture Design document

UPEDU- Unified Process for Education; software development process that streamlines development

str- string

cin- standard input stream

Unix- a family of command-based operating systems which differ from macOS and Windows

Linux- a Unix operating system

1.4 References

Saiedian, Hossein. (2024). Software requirements engineering [PowerPoint slides]. EECS 348 Fall 2024 Canvas.

Saiedian, Hossein. (2024). Striving for Successful Team Projects [PowerPoint slides]. EECS 348 Fall 2024 Canvas.

1.5 Overview

Architectural Representation - Describes the software architecture for the system, how it's represented, enumerates the views that are necessary, and explains the types of model elements for each view.

Architectural Goals and Constraints - Describes the software requirements and objectives that have a significant impact on the software architecture. This can include safety, security, privacy, use of an off-the-shelf product, portability, distribution, and reuse. This section also discusses the special constraints, such as design and implementation strategy, development tools, team structure, schedule, legacy code, etc.

Use-Case View - Describes use cases or scenarios from the use-case model if they represent some significant functionality of the final system, or if they have a large architectural coverage.

Logical View - Describes the architecturally significant parts of the design model. It introduces architecturally significant classes and describes their responsibilities, as well as important relationships, operations, and attributes.

Interface Description - Describes the major entity interfaces. This includes screen formats, valid inputs, and resulting outputs.

Size and Performance - Describes the major dimensioning characteristics of the software that impact the architecture, as well as the target performance constraints.

Quality - Describes how the software architecture contributes to all capabilities of the system. This includes extensibility, reliability, portability, etc.

2. Architectural Representation

Architecture for the *Arithmetic Expression Evaluator* (AEE) will mostly be split up between files and functions that will each fill out separate mathematical expressions and roles. There are two ways to go about organizing the architecture of the AEE:

1. Each use case (parenthesis, exponent, multiplication and division, etc) will have its own C++ class, and each class file will consist of a main and supporting functions (if needed). There will be one final file that takes input and calls other files and functions based on the details of the input. This file will also catch and smooth out any errors that other classes raise. For example, if the input includes multiplication outside of parentheses and addition inside the parenthesis, then the final file will call the file for addition, then parenthesis, then multiplication.
   The supporting functions within each class, when needed, will organize code into cohesive blocks, making the entirety of the code more readable and efficient. Supporting functions will include any code that is called multiple times, or code blocks that serve a separate functionality from a majority of the other code in the main function of the C++ file.
2. Each use case will have its own function, and each function will have supporting functions if the main use case function becomes extensive or complex. Supporting functions will be necessary to organize code into more manageable blocks if code is needed more than once, becomes extensive, or serves a separate (supporting) functionality from the main use case function code.
   The main function of the file will take input and process it before calling other functions to fill out each use case. It may still be necessary to have multiple files for this scenario to avoid making one file too large, but if there are multiple files, they will be grouped based on use case. For example, parenthesis would have its own file, exponentiation and multiplication and division would also have its own file, and the main file would take in input and call other files, as well as deal with any errors that are raised.

The choice between these two options will be made based on the requirements of complexity later specified in the implementation phase by the EECS 348 professor. If lots of work, detail, and exertion is needed, then the first option for architecture will be necessary. However, if the complexity for the AEE is to be more minor, then the project will progress with the second option, which will be more time & resource effective.

3. Architectural Goals and Constraints

Functional requirements are the biggest dictator for architecture. Since complexity and extensiveness of implementation has not yet been defined, there are two scenarios for the architecture outlined in section 5 of this document that are shaped by functional requirements. In short, each functional use case will either have its own function or its own file.

Nonfunctional requirements and constraints also require code to be organized into either scenario to keep with requirements of safety and privacy, readability, reuse, and compilation. Safety and privacy play into architecture by keeping all variables and objects local to functions or files, and ensuring that any human non-human users cannot interact with these variables and objects to avoid mistakes that could be made by the user if they lack understanding of the *Arithmetic Expression Evaluator* (AEE). Separating code into functions and files also facilitates readability so that other entities can easily comprehend the AEE. Reuse of code arises often within software so having a separate function or file for each use case facilitates reuse and decreases the amount of processing power and memory the AEE will take up on a device. Having a separate file or function for each use case also ensures the compiler has an easier time running input through all the possible functions and files that could take up space on the stack or heap during runtime.

Overall, requirements, whether functional or nonfunctional, structure the way that the implementation of the *Arithmetic Expression Evaluator*.

4. Use-Case View

4.1 Parenthesis

Use-Case Flow

Primary Flow:

1. AEE checks validity of mathematical expression containing parenthesis
2. AEE determines which statements have increased precedence due to parentheses
3. Any operations with increased precedence due to parentheses are called

Exceptional Flow:

1. AEE finds the expression containing parenthesis is syntactically incorrect; invalid input use case called

Example usage:

1. User inputs the string "(9+10)*5"
2. AEE finds opening and closing parentheses in the string
3. AEE elevates the precedence of "9+10" above all other expressions
4. Addition use case is called
5. Multiplication use case is called
6. All terms have been evaluated; AEE outputs the evaluated expression

4.2 Exponentiation

Use-Case Flow

Primary Flow:

1. The user enters a mathematical expression into the terminal.
2. The AEE scans for expressions that contain exponents.
3. The AEE looks for parentheses that need to be calculated before the exponent.
4. If the AEE finds any parentheses, it will call the parentheses use case.
5. The Evaluated result is substituted back into the original expression.
6. The exponent is computed by using a loop to multiply the base number by itself, since C++ does not support exponentiation directly.
7. The final calculated result is returned and displayed in the terminal, along with anything from other use cases.

Exceptional Flow:

1. If the AEE detects improper input, it rejects it and calls the error handling use case.
2. The error message function is called and prints error messages to the user.

Example usage:

1. A user enters the expression: (2 + 4) ^ 2.
2. The AEE identifies the parentheses around 2 + 4.
3. The addition use case evaluates 2 + 4 and returns 6.
4. The original expression is updated to 6 ^ 2.
5. The exponentiation use case calculates 6 ^ 2.
6. The final result, 36 is displayed to the user.
7. The program returns to main and waits for other function calls.

4.3 Multiplication and Division

Primary flow:

1. AEE reads input from user
2. AEE breaks down input into smaller expressions or encounters syntax error
3. Evaluates any possible parentheses or other operators with higher precedence by handing over to its respective use case
4. The AEE computes the multiplication or division by using built-in operators provided by C++
5. Final calculated result is returned and displayed in the terminal

Exceptional flow:

1. If the AEE detects any errors it rejects the input, this includes unmatched parentheses, syntax errors, mathematical errors (divide by zero), and non-numerical input
2. Error message function is called to display an error message to the user

4.4 Modulus

Use-Case Flow

Primary Flow:

1. The AEE reads an input from the user.
2. The AEE breaks down the input into smaller expressions or encounters a syntax error.
3. Any possible use cases that take precedent over Modulus are evaluated.
4. The AEE computes the modulus by taking whatever proceeds the “%” modulus whatever follows the “%”.
5. The final calculated result is returned and displayed in the terminal.

Exceptional Flow:

1. If the AEE detects any improper input, it rejects the input.
2. The error message function is called and displays an error message to the user.

4.5 Addition and Subtraction

Primary Flow

- User Enters Expression: A human user inputs a mathematical expression with addition and/or subtraction operators.
- System Parses Expression: The AEE breaks down the expression to identify + and - operations and operands.
- Check for Higher Precedence Operations: The system evaluates higher-precedence operations (like parentheses or multiplication) first, if present.
- Evaluate Addition and Subtraction: Once all other operations are resolved, the system performs addition and subtraction operations from left to right.
- Output Result: The AEE displays the evaluated result or returns it to the calling system component.

Exceptional Flow:

- Invalid Syntax: If the system identifies invalid syntax (e.g., consecutive operators), it halts evaluation, provides an error message, and prompts the user for corrected input.
- Non-Numeric Input: If the expression includes non-numeric values (other than valid operators), the system returns an error indicating invalid input.
- Overflow/Underflow: If the addition or subtraction exceeds data type limits, the system warns of an overflow or underflow issue.
- Unmatched Parentheses: If parentheses are unmatched in the expression, an error message prompts the user to correct the input.

5. Logical View

Since there are two main architecture scenarios, each option will be discussed in the subsection. The first scenario where each use case has its own file will be referred to the File Case. The second scenario where each use case has its own function will be referred to as the Function Case. In Section 5, these two scenarios will be used in a generalized manner to describe the individual pieces of architecture.

In both of these scenarios, the AEE architecture will be constructed entirely from smaller modular subsystems. One type of these subsystems is the operations subsystems, these will manage basic math operations, including multiplication, division, addition, and subtraction. These operations will be self-contained, minimizing unnecessary dependencies between them. To do this, the subsystems will return generic types that can be used directly by separate functions without modification. This ensures that when an operation uses values returned from another, it can treat the return values as if they were input received directly from the terminal.

One of the AEE's most essential relationships is between the input processor and the math operation subsystems. The input processor will take the entire expression received from the terminal and decompose it into smaller, manageable parts. These smaller expressions can then be passed into the operations to be evaluated into numbers. This approach promotes a more modular structure and prevents the operation systems from exceeding their intended scope.

5.1 Overview

Basic structure was discussed in Section 2, Architectural Representation, where each use case has either its own class in its own file, each with at least one function but often multiple, depending on required complexity during the implementation phase.

Given this, there also needs to be a file that runs each of these use cases. Such will be the main file, which will take in input through its main function, process the input in a separate class, then return back to the main function to make decisions on appropriate calls to other use cases. The main file will be the first and last one to run, and will be the first item on the stack and last one off the stack. It is in charge of managing the entirety of how the *Arithmetic Expression Evaluator* (AEE) runs and is at the top of the hierarchy within the AEE project.

Since input is processed first, and has the biggest influence on what calls main will make after its return, the structure that processes input is second in hierarchy. It is very important that this use case properly processes input, otherwise, the rest of the evaluations and processes in the AEE will be evaluated incorrectly or will raise errors. Because the input use case holds such high importance, it is placed second in hierarchy and is the second layer to the *Arithmetic Expression Evaluator*.

The next in hierarchy will be the use case of parenthesis, since this will indicate absolute priority in the line of input, as per basic mathematical priority concepts. Parenthesis will be identified from the line of input in their own class, and this class and its function will return information on what is to be evaluated inside the parenthesis back to main. This means that the parenthesis use case influences what calls main will make, and thus is third in the hierarchy of the AEE.

All other evaluatory use cases are last in hierarchy and are the bottom, but still essential, layer to the AEE. This includes exponentiation, multiplication and division, modulus, and addition and subtraction. This is because these use cases do the work of evaluating the expression and do not play a significant role in decision-making based on input, and do not do any significant processing of the input other than of floats and integers. These use cases are still important and hold together the platform that the *Arithmetic Expression Evaluator* sits on to be a complete and effective project.

5.2 Architecturally Significant Design Modules or Packages

The Arithmetic Expression Evaluator (AEE) is organized into packages to construct key functionalities of the evaluator. Each package is dedicated to handling specific mathematical operations to facilitate reusability, maintainability, and efficient computation. Below is a description of each package and its components:

Name: Main
Description: This package takes in an input string from the human or non-human user and makes calls and main decisions based on returns of other calls in the hierarchy.

Classes

- Main(): Responsible for taking in input and responsible for making decisions for various calls to other functions and files.
- Responsibilities
    - Take in string of input from human or non-human user
    - Send input string to Input Processor and send return of that to Parenthesis Handler
    - Make decisions on which function/file calls to make based on return from Input Processor and Parenthesis Handler
- Operations:
    - main(): handles all above responsibilities

Name: Input Processor
Description: This module is called by Main to process the string input, parse it, check validity, and return the user input in an appropriate format to Main.

Classes

- inputProcessor()
- Responsibilities
    - Taking user input from the standard input stream (cin) and converting that to a string
    - Checking input for validity
    - Cleaning input
- Operations:
    - getInput(): Gets input from cin; calls cleanInput; calls checkInput. If checkInput returns a -1, getInput will inform Main to terminate the program. If checkInput returns a 0, getInput will return the cleaned, checked string. If checkInput returns a 1, getInput will print an error message to the terminal.
    - cleanInput(): Cleans up an input string by removing whitespace within the string as well as leading zeros (eg. converts ‘5 + 012’ to ‘5+12’). Returns the cleaned string.
    - checkInput(): Checks input for validity. If the user inputted a ‘q’ to quit, checkInput will return a -1. If two mathematical operators (excepting two parentheses or two asterisks) are placed side-by-side or if non-numerical, non-operator characters are present, checkInput will return 1. Otherwise, it will return 0.
        - Note: errors within mathematical expressions, such as dividing by zero, will be evaluated by the appropriate mathematical function module. checkInput will only check the validity of syntax.

Name: Parentheses Handler
Description: This package processes expressions within parentheses, giving precedence for calculations.

Classes

- evalParenthesis(): Responsible for parsing and evaluating sub-expressions within parentheses
- Responsibilities
  - Locate matching pairs of parentheses in expressions.
  - Recursively evaluate inner expressions to ensure correct order of operations
- Operations:
  - findParentheses(): Finds indices of matching parentheses.
  - evaluateSubExpression(): Identifies and marks the expression within a given pair of parentheses that Main can identify

Name: Exponentiation
Description: This package finds the exponentiation operator and identifies base and exponent to evaluate.

Classes

- exponentiation(): Responsible for identifying the base and exponent of an expression that is passed to it, then evaluating it and returning it.
- Responsibilities
  - Locate exponent operator in expression.
  - Identifying the base to the left of the operator.
  - Identifying the exponent to the right of the operator.
  - Using a for loop to multiply the base by itself n amount of times where n is the value of the exponent identified.
- Operations:
  - identifyComponents(): identifies and marks operator, base, and exponent to variables.
  - evlExponent(): contains for loop that multiplies the base exponent amount of times

Name: Multiplication and Division
Description: This package takes in an expression that contains a multiplication or division operator, then calculates the value for the expression. This value is then returned.

Classes

- multDiv(): Responsible for parsing and evaluating sub-expressions within parentheses or hanging operands and operators.
- Responsibilities
  - Finding multiplication/division operators inside of an expression.
  - Enforce rules regarding multiplication/division. Returns error if the user divides by zero.
- Operations:
  - validateInput() checks for errors in expression and cleans input in preparation for sending it to an evaluation function (evaluateMult() or evaluateDiv())
  - when validateInput() is done running, it checks whether to be passed off to evaluateMult() or evaluateDiv()
  - evaluateMult() will evaluate the product of operands on the sides of the operator
  - evaluateDiv() will evaluate the quotient of the number on the left of the division operator by the number on the right of the division operator.

Name: Modulus
Description: This package takes in an expression that contains a modulus operator, then calculates the numerical value for the expression. This value is then returned to Main.

Classes

- modulus(): Responsible for handling expressions that contain the modulus (%) operator.
- Responsibilities
    - Detecting a modulus symbol inside of a given expression.
    - Enforce mathematical rules pertaining to modulus operations. For example, modulus should throw an error if the operation is between two floats.
    - Additional processing of input will be required if the expression contains a negative number in the dividend. This is due to the fact that C++ does not correctly handle negative dividend modulus operations.
- Operations:
    - evaluateModulus(): Takes in a modulus expression and returns an integer as the result.
    - validateInput(): Checks for errors in the given expression, then cleans the input in preparation for sending it to evaluateModulus().

Name: Addition and Subtraction
Description: This package processes expressions that have addition and subtraction operators and evaluates them, returning the result to Main.

Classes

- addSub(): Responsible for parsing and evaluating sub-expressions within parentheses or hanging operands and operators.
- Responsibilities
    - Locate addition and/or subtraction operators in expressions.
    - Evaluate the expressions left to right by taking the operand to the immediate left and to the right of the operator and solving them, then moving onto the next left-most operator.
- Operations:
    - findOperators(): finds operators from left to right and calls add() or sub() based on if its add or subtract, sending the operators to the immediate left and right of operator
    - add(): evaluates expression by adding two given operands
    - sub(): evaluates expression by adding the negative of the second operand to the first

6. Interface Description

Non-Human User Interface

The user interface for non-human users, to be used in a larger system, will be very concise and simplified so that the larger system can just take the return of *Arithmetic Expression Evaluator* (AEE) and use it. This interface will be limited to request for input of expression and then return the answer evaluation, without any other text such as “result” or “answer.”

Example Messaging
enter expression: 3(5+2)
21

Human User Interface

The human user interface for the AEE will be designed to be simple and readable, with emphasis on clarity of instructions to the user. The first iteration of the user interface will be limited to input and output from the command line interface. If time allows, further iterations of the user interface will beautify the interface, since the command line can be confusing for computer-illiterate users. These iterations may include building a graphical user interface application or creating a web-based application. Due to time and programming language constraints, however, the user interface will likely be limited to the command line, so efforts will be made to beautify the command line and facilitate easy use for all possible users.

Upon running the AEE, the command line will display a simple welcome message and prompt the user to input a mathematical expression. Subsequent versions of the AEE may also allow the user to request examples of valid/invalid input, request explanations of operators, or to change the displayed language of the AEE. However, due to time constraints, it is likely the user interface will only display simple prompting and welcome/goodbye messaging. The user interface will also display error messaging if errors in the user input are found, or if there are other failures of interpreting input. Such errors will be clear and concise. For example, if a user attempts to divide by zero, the error may be displayed in a user-friendly message (“Undefined expression- user attempted to divide by zero”) or as a Python-esque error message (“ZeroDivisionError”).

Example Messaging
“Welcome to the Arithmetic Expression Evaluator!
Please input a mathematical expression or ‘q’ to quit: ”
“Goodbye!”

Valid Inputs and Resulting Outputs
Any possible combination of ASCII characters is considered valid input for the AEE; however, if the input is not a valid mathematical expression, the AEE will display an “Invalid input” message.

Input- 51 + (70-5)
Output- 116

Input- 43a7 + 4
Output- Invalid input

Input- 47/0
Output- Undefined

Input- 5 % 6 + 5**2
Output- 30

7. Size and Performance

Scalability:

Scalability relates to how easily a system can handle a variety of different sizes, or scales, of user inputs. The AEE will be capable of handling arithmetic expressions of all sizes in a smooth and timely manner, and will not struggle with a higher workload. This is achieved by making each operator into separate functions. The AEE will not need to struggle with containing every operator all together, and will in turn run more efficiently.

Concurrency:

Concurrency is the ability for a system to take on and handle multiple requests simultaneously. The AEE will more often than not be taking on several requests at a time, and it will be prepared to handle all of them. For example, a user may input the equation "(2 + 2) * 5". The AEE will read this equation and find that it both needs to deal with the parentheses and the contents inside, as well as the multiplication that follows. The AEE will handle all of these in an efficient manner in the order of operations, and return the result to the user.

Data Volume:

Data Volume refers to the amount of data the system needs to store and process. In the case of the AEE, the data that it will need to contain is any input equations from the user. The AEE will take this data, analyze which operators are needed to solve the equation, and return the result to the user, as described before. Alternatively, if there is a syntactical error within the equation, the AEE will detect this and return an error to the user. The AEE will also be capable of remembering and using solutions to parts of an equation. For example, take the equation "(2 + 2) * 5". By the order of operations, "(2 + 2)" will be the first part of the equation solved, resulting in 4, but the AEE will retain this information and use it to multiply by 5 later on.

Response Time:

Response Time refers to how long the system takes to respond to a user request. For the AEE, this will be the time it takes between the user inputting an equation, and the system outputting its solution. This process will be quick and efficient. The AEE will read inputted equation, quickly determine which operators are needed and in what order, and run the prompted functions. Each function will be programmed to run smoothly and efficiently, there will not be any wasted time calculating an answer. Of course, longer equations will take more time, but that will be nearly negligible.

Maintainability:

Maintainability is how easily the software can be modified or changed over time, either to keep it up to date or to allow it to run better. For the AEE, this could theoretically be needed when it is used within a larger system. Requests might be put in to streamline the AEE to allow faster output, or for answers to be formatted in a different way. The program will be easily readable and readily modifiable to those who have access, as each function will have several comments indicating what it does. It is also expected that there will be several iterations of the AEE before the product is complete.

8. Quality

The architecture of the Arithmetic Expression Evaluator (AEE) is designed to ensure that the system meets high standards of the required elements such as: Extensibility, Reliability and Portability. Each element will be discussed furtherly within this section, as well as key quality attributes that will contribute to evaluators overall strength.

Extensibility:

The AEE will make sure to enable easy addition of new features such as complex mathematical operations without major revision. To achieve this, each operation will be organized into separate, self-contained packages/classes, ensuring user-friendly/smooth results for clients. This includes operations like Error Handling, where the objective is for the AEE to automatically handle errors within appropriate timing/setting as well as correct error outputs. For example, if a client was to enter 2 +* 2, the error output would result as whatever the client wishes, which may be "Operator overloading, operation must either contain parenthesis or one of the arithmetic symbols". If a client was wanting to add additional features such as logarithms or trigonometry, the AEE would allow the client to do this by simply creating new modules/classes for these operations. Adding new functions to the evaluator should be both easy and accurate, as clients deserve to get what they want/need.

Reliability:

The AEE will ensure reliability as it is considered a priority. The AEE will be implemented to handle user input accurately and prevent system crashes which can be caused by either invalid/complex expressions inputted. Operations will include error-handling functions that will provide feedback regarding issues found in the input (i.e. syntax errors, dividing by zero, etc.). The input processor module will verify if the input is valid before proceeding to run the code, which is a great way to save time and avoid the system encountering errors later on. The AEE also ensures that functions are kept separate from one another; in other words, each function will be independent which will avoid errors happening in more than one function.

Portability:

Portability is highly valuable, as it will help the AEE become accessible throughout many different platforms with minimal adjustments being made. By simply relying on C++ libraries, rather than platform-specific libraries, the AEE will be able to operate within nearly any environment. The AEE will be composed of separate modules which will each contain its own feature (i.e. Addition, Exponentiation, Multiplication, etc.). Each module will be able to handle one function, and each module will not share the responsibility on separate modules. The AEE will also be able to be tested on multiple platforms to confirm its portability; such platforms include Windows, Linux/Unix, MacOS, and many more.

Privacy and Security:

The AEE must not only be easy and accessible for clients, but also must make clients feel safe and protected. While it is true that the AEE will not handle any sort of sensitive data, it includes secure coding to avoid exposure to anything/anyone outside the client's own AEE (particularly input validation and error handling). As mentioned previously, the AEE includes an input processor module which has the ability to remove any potentially harmful characters/sequences that may crash the system. To avoid external manipulation, variables and functions are kept local within modules which ensures the AEE will operate/behave correctly within expected parameters. Most importantly, data processed by the AEE will not be stored after the processing execution, which benefits for security purposes.

Maintainability:

Perhaps the most unrecognized part of the AEE is its maintainability. Through maintainability, which can be achieved through understandable documentation, well-structured code, and a module design that will simplify updates/debugging, the AEE will only be beneficial for clients. Each mathematical operation will have its own separate function/class, making it easier to locate and modify code. Code will also have the option to include comments for better understanding/logic. The AEE's error-handling mechanism will make it easier to fix issues across modules.

In summary, with the following qualities above, the AEE will ensure the system remains adaptable and efficient while delivering precise results across a range of various arithmetic scenarios. The design approach of the AEE allows easy addition of new features for the benefit of the user's experience and the system's strength.