Kansas Instruments™ (not really)

Arithmetic Expression Evaluator in C++
Software Architecture Document

Version 1.4

Revision History

Date	Version	Description	Author
10/31/2023	1.0	Overview of the SAD, work on section 1 began and assigned to be done by Friday.	Benjamin Kozlowski, Nicholas Hausler, Steve Gan, Jacob Leehy, MJ McGee
11/03/2023	1.1	Work began in earnest on other parts of the SAD, decided on function-oriented design.	Benjamin Kozlowski, Nicholas Hausler, Steve Gan, Jacob Leehy
11/07/2023	1.2	Worked on sections of the SAD, began focus on the Logical view and diagrams	Benjamin Kozlowski, Jacob Leehy, MJ McGee, Steve Gan, Nicholas Hausler
11/10/2023	1.3	Final review before publishing on GitHub	Benjamin Kozlowski, Nicholas Hausler, Steve Gan, Jacob Leehy, MJ McGee
12/1/2023	1.4	Added final touches to references and small corrections, changed package diagrams	Benjamin Kozlowski, Nicholas Hausler, Steve Gan, Jacob Leehy, MJ McGee

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

The purpose of the Software Architecture Document is to provide information on the architectural organization, how the software is decomposed into components, and the technologies that were used to build the software. This is done in order to ensure performance, usability, security, reliability and maintainability. This document also communicates our architectural design process and decision making to any entity that may need to review the project documentation and determine our implementation rationale.

1.2 Scope

This Software Architecture Document applies to the software architecture required/used towards creating an arithmetic-based calculator. This document applies and influences the software architectural quality, the use-case parameters used in the arithmetic-based calculator, and the represented software architecture created and used towards the project.

1.3 Definitions, Acronyms, and Abbreviations

EECS – Electrical Engineering and Computer Science

KU – University of Kansas

SA – Software Architecture

SAD – Software Architecture Document (This document)

SRS – Software Requirements Specifications

UML – Unified Modeling Language

1.4 References

GitHub Repository: The project will be documented through a GitHub repository containing all relevant information regarding the project, including the Software Development Plan and this file.

Software Development Plan (Document 1): The plan for the project including the organization, jobs, focuses, scope, deliverables, and general overview of the project.

Software Requirements Specifications (Document 2): The document denoting the requirements, both functional and non-functional, for the project.

User Manual (Document 4): This document will go over the steps on how to get started with the program, what it can and cannot do, and give examples on how to correctly use the Arithmetic Expression Evaluator in C++.

Test Cases Document (Document 5): This document houses the multitude of test cases that were created to attempt to find and fix any bugs or problems with the program.

Meeting logs: Notes taken by the Recording/Secretarial Engineer describing the activities within each meeting and other miscellaneous information.

Glossary:

- EECS: Electrical Engineering and Computer Science
- KU: The University of Kansas
- UML: Unified Modeling Language

1.5 Overview

The following sections of the SAD contain the architectural representation of the project, the goals and constraints for the architecture, a logical view of the significant parts of the design model, a description of the user interface, and other quality capabilities of the architectural design. It is organized in this manner to give a clear understanding of why the representation was chosen, a logical view of it, and how it will affect other qualities and interfaces.

2. Architectural Representation

Our current intention for the system is to develop it using a function-based architecture, with each function being called on a case-by-case basis as needed within the evaluation process. These functions will be loosely grouped into categories based on their functionality. Functional architectures tend to apply effectively to simpler projects and given the relatively small scope of our calculator, it is rational to follow a functional paradigm for this project. The functions within the system will be grouped into 5 categories:

- Stack Operations: All standard stack functionalities such as push, pop, top, etc. These will be used by the parser to queue expressions for calculation.
- Parsing: Splits the expression into manageable expressions, following the correct order of operations as defined in the SRS.
- Evaluation: Pops expressions from the stack and calls operations as needed to calculate the final answer.
- Mathematical Operations: Handles addition, subtraction, multiplication, etc. This group of functions will take the parsed functions and evaluate them.
- Outward Functionality (main): Handles input, error handling, responding to the user, etc. This group of functions will be used to get information from and communicate the output to the user, as well as inform them of any errors they made with their input.

3. Architectural Goals and Constraints

For this project in particular, constraints such as security and privacy should not be an issue of much concern, if any. Other goals and constraints that our project will need to consider are as follows:

Goals:

- Distribution: All distribution of the implementation will be handled through GitHub and will operate on an individual need basis.
- Reuse and Decomposition: The program contains classes/functions that allow for reusability of arithmetic functions and are broken down such that they may be used in a way which provides versatility.
- Legacy Code: The program files and their iterations will be saved on GitHub and can be accessed at any time. When the deadline is reached by the end of the 2023 Fall Semester, the code will be accessible on sources that allow C programming code and will not be updated, reaching its final update.
- Use of off-the-shelf products: The program will use the standard I/O, string, stack, and the math libraries in C++.

Constraints:

- Safety: The program cannot enter any infinite loops or recursion, and errors will be handled without the program crashing.
- Development Tools: The program is limited by the constraints of C++ and its libraries.

- Team Structure: As our team is only comprised of five members, each will be responsible for a relatively large portion of the work on the program, and this work will be divided to ensure quality and effort. Each member will also continue to perform their individually assigned roles as determined in the Software Development Plan.
- Schedule: The program implementation, as well as all the Software Development Documents are required to be done by the end of the 2023 Fall Semester.

4. Use-Case View NA

4.1 Use-Case Realizations NA

5. Logical View

5.1 Overview

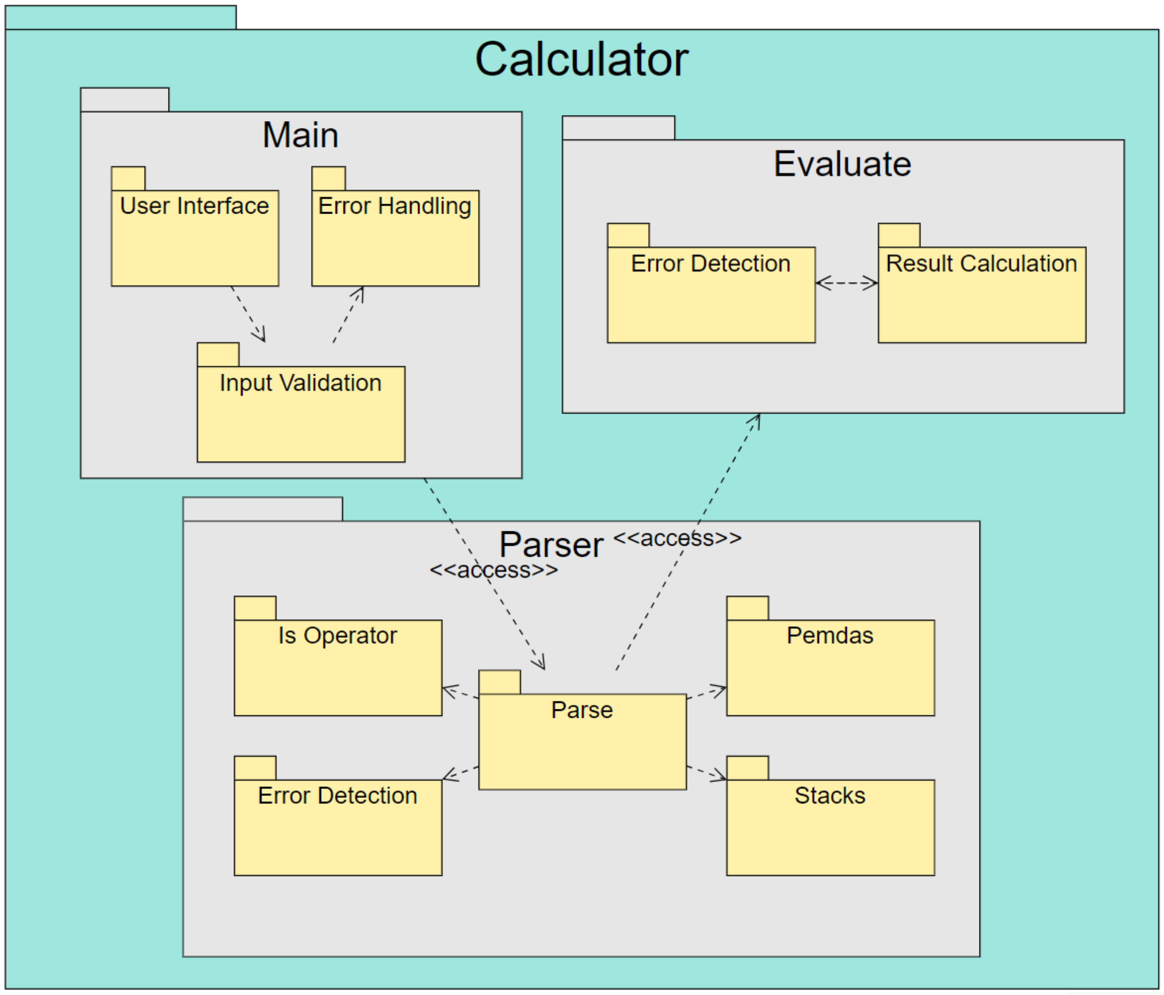
The functions that comprise the system will be composed as such:

- Main: This will proceed through the following process
  - Take in the user input
  - Call CheckInput to verify input
    - This will either continue the program or require reinput
  - Call the parser with the inputted string
    - Any errors generated by this will be handled by main
  - Receive the calculated answer from the parser.
  - Display the final answer
- CheckInput: This will take a string and will iterate over the input string to verify that the string structure is acceptable. Then, it will raise an error if needed.
- Parse: This function will take a string as a parameter and parse it into understandable pieces that the program can handle.
  - It will check for grouping symbols first, breaking up the expression as needed. Then it will break down the equation according to the order of operations.
  - Parse will push the broken-down values to the stack.
  - After the decomposition, parse will call Evaluate.
  - Parse will return the result of evaluate to main.
- Stack: The stack will be string based and will accept string objects based on the string class. The implementation will be done using the built in C++ stack module, hence why the inner workings will not be displayed in later diagrams.
  - Push: Pushes a string to the top of the stack. The string will be passed as a parameter.
  - Pop: Removes a string from the top of the stack. This string is returned.
  - Top: Returns the string currently at the top of the stack.
  - IsEmpty: Returns a boolean value based on if the stack is currently empty.

- Evaluate: This will pull items from the top of the stack using pop. It will then call math operations as needed to simplify and combine the expression. This will return the final answer to parse once the stack is empty.
    - Addition: takes in two ints as parameters and returns the sum of the two ints.
    - Subtraction: takes in two ints, x and y, as parameters and returns the result of subtracting y from x
    - Multiplication: takes in two ints as parameters and returns the product of the two.
    - Division: Takes in two ints, x and y, as parameters and returns the result of dividing x by y. Trying to divide by 0 will result in the program returning an error.
    - Modulo: Takes in two ints, x and y, as parameters and returns the result of x modulo y.
    - Exponent: Takes in two ints, x and y, as parameters and returns the result of x to the yth power.
- Output: This will be handled by main and will function as a single print to console statement.

5.2 Architecturally Significant Design Modules or Packages

The following section will give a brief description of the significant portion of the program, as well as a function diagram showing main functionalities and interaction with other portions of the program.



Main:

- Implements the user interface
- Validates the input
- Handles errors raised by the program
- Calls Parse on the input
- Displays the final answer to the user

Parser:

- Takes the input from main
- Parses the string by pushing pieces of information to the stacks
    - Detects errors in the string while parsing
- Uses isOperator and Pemdas to determine the methodology needed to correctly evaluate the expression
- Calls evaluate on the broken-down chunks of the expression

- Returns the final answer to main

Evaluate:

- Takes information from the parser
- Determines the needed operation and evaluates the chunk of data
    - Detects errors in the expression before calculation
- Returns the calculated values to the parser

6. Interface Description

The main interface that will be interacted with regarding the program is the command line in the terminal. From the command line, the user is prompted to enter an expression into the terminal and the terminal will then either output an error message if any errors in the input were detected, or it will output the correct solution to the entered expression.

Valid expressions should contain only the following: numerical characters, supported operands (+, -, *, /, %, ^), and grouping symbols (parentheses). The user can also type ‘q’ to quit the program. Grouping symbols should be properly matched, but extraneous sets of grouping symbols are fine. If the input is determined to be valid, the program will output the result of the expression. Otherwise, the program will give the user a descriptive error message based on which error it found. After the program produces an output based on the user’s expression, the user is then again prompted to enter another expression to be evaluated or the user can type ‘q’ to quit the program.

7. Size and Performance NA

8. Quality

- Comments: Source code should be readable and understandable to any outside programmer.
- Error Handling: The software should be able to identify invalid or undefined behaviors and safely inform the user instead of crashing out or freezing.
    - Safety: The software should never allow the user to perform operations or computations that could result in memory leaks or data corruption and should handle errors appropriately.
- Generalization: The code should be well defined so that it does not get repeated and so functions are used several times.
- Decomposition: Components for the larger system should also be built from finer-grain elements.
- Privacy/Security: The program will never save or ask for personal information from the user and the program will not be connected to the internet.