Kansas Instruments

KI-69 Calculator
Software Architecture Document

Version <1.0>

[Note: The following template is provided for use with the Unified Process for EDUcation. Text enclosed in square brackets and displayed in blue italics (style=InfoBlue) is included to provide guidance to the author and should be deleted before publishing the document. A paragraph entered following this style will automatically be set to normal (style=Body Text).]

[To customize automatic fields in Microsoft Word (which display a gray background when selected), select File>Properties and replace the Title, Subject and Company fields with the appropriate information for this document. After closing the dialog, automatic fields may be updated throughout the document by selecting Edit>Select All (or Ctrl-A) and pressing F9, or simply click on the field and press F9. This must be done separately for Headers and Footers. Alt-F9 will toggle between displaying the field names and the field contents. See Word help for more information on working with fields.] Marked (shaded) areas: items that are OK to leave out.

Revision History

Date	Version	Description	Author
<dd/mmm/yy>	<x.x>	<details>	<name>
9/11/2023	1.0	Added information to sections 1.3-1.5	Jenna Luong
10/11/2023	1.1	Started on all sections	Jenna Luong, Nikka Vuong, Hayden Roy, Harrison Wendt, Dylan Sailors, Ginny Ke
12/11/2023	1.2	Added information to section 2, filled Out Section 6 and 8, & finalized information for the rest of the sections	Ginny Ke, Hayden Roy, Jenna Luong, Nikka Vuong, Harrison Wendt, Dylan Sailors

Table of Contents

Software Architecture Document

1. Introduction

This document will provide a detailed architecture design of the Arithmetic Expression Evaluator in C++ program that will be implemented in this project. The purpose, scope, definitions, references, and overview are all detailed below.

1.1 Purpose

The purpose of a Software Architecture Document is to identify and describe both functional and non-functional requirements such as organization & maintainability, scalability & adaptability, user-friendliness, and reusability.

1.2 Scope

This Software Architecture Document provides an overview of the architectural goals, constraints, interface descriptions, and quality that will be used to create the Arithmetic Expression Evaluator in C++.

1.3 Definitions, Acronyms, and Abbreviations

PEMDAS:

- Order of operations. Parentheses, Exponent, Multiplication, Division, Addition, and Subtraction.

C++:

- General-purpose programming and coding language. This is the language that will be used to create our program.

Arithmetic Expressions:

- +, -, /, *, %, ^, and numeric constants.

UML:

- Unified Modeling Language, a standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers for specifying, visualizing, constructing, and documenting the artifacts of software systems.

1.4 References

Refer to the Appendix.

1.5 Overview

In the following sections, the software architecture is described. The architectural representation, goals, and constraints are detailed in sections 2 and 3 respectively. The logical view of the software architecture will be described in section 5, and section 6 will contain the interface description. Lastly, the quality of the software architecture will be defined in section 8.

2. Architectural Representation

This document presents the architecture through a logical view which can be found in section 5. These layered views are shown through the UML model developed on visual paradigms. The model elements for the logical view include the operators, input value, and derived value.

Operators:

- Accurately identify and execute the given arithmetic expressions.

Input Value:

- Identify the data given by the user and accurately group expressions when needed.

Derived Value:

- Output the results of the given input data following the rules of PEMDAS.

3. Architectural Goals and Constraints

Architectural Goals:

- Functionality: Program should be able to accurately parse arithmetic expressions and calculate results following PEMDAS.
    - Should handle different arithmetic expressions.
- Organization and Maintainability: Codebase should be well-organized, modular, and documented to maintain organization and make future enhancement of the codebase easier.
- Scalability and Adaptability: System should be designed to handle an increase in supported features without losing performance or maintainability.
- User-Friendly: Interface should make program easy to use for the end user. This includes an easy way to get user input with a clean output.
- Reusability: Program should be capable of being seamlessly integrated into other programs.
- Test Cases: Develop test cases to verify the correctness of the expression evaluator and to identify bugs in the program so that they can be corrected.

Architectural Constraint:

- Design and Implementation Strategy
    - Implement calculator program using C++.
    - Use object-oriented programming principles to structure code.
    - Program should appropriately handle errors in expressions.
    - Program should evaluate expressions according to the order of operations.
- Schedule
    - Software Architecture by 11/12
    - Implementation by 12/3
    - Test Cases by 12/3
    - User Manual by 12/3
- Team Structure
    - Team Administrator/Lead: Dylan Sailors
    - Assistant Team Administrator: Ginny Ke
    - Project Leader: Harrison Wendt
    - Assistant Project Leader: Hayden Roy
    - Technical Leader: Jenna Luong
    - Data Administrator/Quality Assurance Engineer: Nikka Vuong

4. Use-Case View

[This section lists use cases or scenarios from the use-case model if they represent some significant, central functionality of the final system, or if they have a large architectural coverage—they exercise many architectural elements or if they stress or illustrate a specific, delicate point of the architecture.]

4.1 Use-Case Realizations

[This section illustrates how the software actually works by giving a few selected use-case (or scenario) realizations and explains how the various design model elements contribute to their functionality. If a Use-Case Realization Document is available, refer to it in this section.]

5. Logical View

This section provides an overview of the architecturally significant parts of the design model for the calculator project. It includes the decomposition into subsystems and packages, introducing significant classes, their responsibilities, relationships, operations, and attributes.

5.1 Overview

Of the architecturally significant parts of the design model, the calculator project is going to be decomposed into 3 main subsystems and packages:

User Interface (UI):
- Responsible for interacting with the user.
- Handles input and output.

Arithmetic Operations:
- Manages arithmetic operations like addition, subtraction, multiplication, and division.

Calculator Utilization:
- Houses the overall calculator functionality.
- Provides supporting classes and functions for the other various operations.

5.2 Architecturally Significant Design Modules or Packages

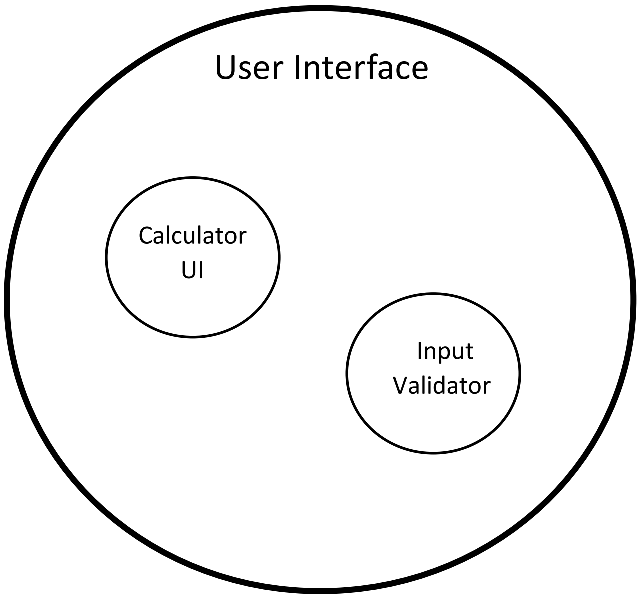
[For each significant package, include a subsection with its name, its brief description, and a diagram with all significant classes and packages contained within the package.

For each significant class in the package, include its name, brief description, and, optionally, a description of some of its major responsibilities, operations, and attributes.]

5.2.1 User Interface

This package is responsible for handling user interactions, input, and output.

User Interface diagram:



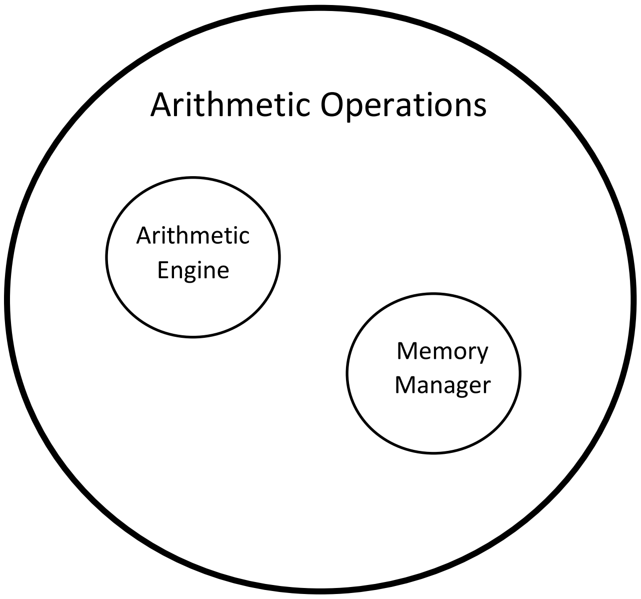
Classes:

- Calculator UI:
  - Description: Manages the user interface components and user input.
  - Responsibilities: Handle user input, display results, and manage the overall user interface flow.
  - Example operations: getUserInput(), displayResult(result)
- Input Validator:
  - Description: Validates user input for mathematical expressions.
  - Responsibilities: Ensure input is syntactically and semantically correct.
  - Example operations: validateInput(input)

5.2.2 Arithmetic Operations

This package contains the core logic for mathematical calculations.

Arithmetic Operations diagram:



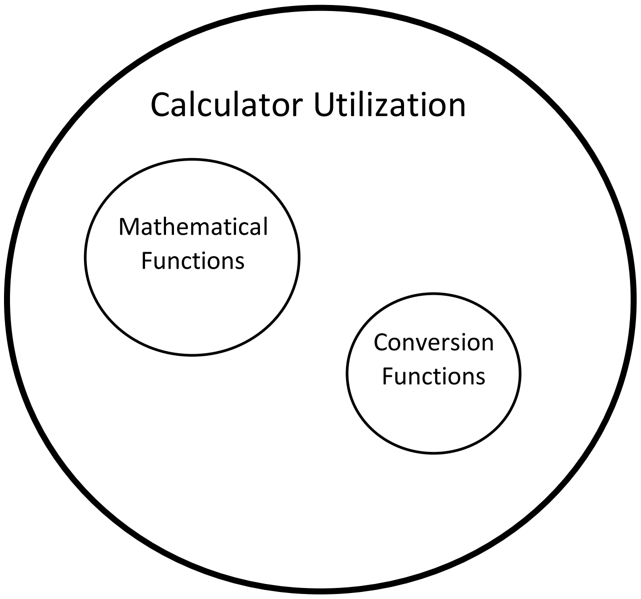
Classes:

- Arithmetic Engine:
    - Description: Processes and goes through mathematical calculations based on user input.
    - Responsibilities: Coordinate operations, manage memory, and perform calculations.
    - Example operations: add(a, b), subtract(a, b), multiply(a, b), divide(a, b)
- Memory Manager:
    - Description: Manages storage and gathering of variables and results.
    - Responsibilities: Store and retrieve user-defined variables and calculation results.
    - Example operations: storeResult(result), retrieveVariable(name)

5.2.3 Calculator Utilization

This package provides supporting classes and functions for various situational operations within the calculator.

Calculator Utilization diagram:



Classes:

- Mathematical Functions:
    - Description: Implements standard mathematical functions.
    - Responsibilities: Provide implementations for functions like sin(x), cos(x), sqrt(x), etc.
    - Example operations: sin(x), cos(x), sqrt(x)
- Conversion Functions:
    - Description: Handles conversions between different units or formats.
    - Responsibilities: Convert between numerical formats (e.g., decimal to binary).
    - Example operations: convertToBinary(decimal), convertToDecimal(binary)

6. Interface Description

Screen Format:

- Text based input and output with input fields for the user to enter mathematical expressions.
- All input and output will be in the terminal.

Valid Inputs:

- Mathematical expressions using PEMDAS operations.
- Users cannot divide by zero.
- User must have an even amount of open and closed parentheses.

Resulting Outputs:

- Result of mathematical expression.
- If input is invalid, return an error message

7. Size and Performance

[A description of the major dimensioning characteristics of the software that impact the architecture, as well as the target performance constraints.]

8. Quality

Extensibility:

- The program should be designed in such a way that components of the program may be extended or replaced with minimal impact on the rest of the program.

Reliability:

- The program should effectively handle errors and have checks within the program to ensure the validity of expressions.
- The testing process should be thorough to ensure that all extraneous input cases are accounted for and properly handled.

Portability:

- The program should be platform independent, allowing it to run on all operating systems.