<Company Name>

QuickMath.io
Software Architecture Document

Version 1

[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>
08/11/23	0.1	Changed introduction	David
11/11/23	0.2	Changed section 8	Vinny
12/11/23	0.3	Finished other sections	Owen, Omar, Jamie, Tatum
12/11/23	1	Cleaned up and uploaded	David, Vinny, Omar, Jamie, Tatum, Owen

Table of Contents

1. Introduction 4
   - 1.1 Purpose 4
   - 1.2 Scope 4
   - 1.3 Definitions, Acronyms, and Abbreviations 4
   - 1.4 References 5
   - 1.5 Overview 5

2. Architectural Representation 5

3. Architectural Goals and Constraints 5

4. Use-Case View 6
   - 4.1 Use-Case Realizations 6

5. Logical View 6
   - 5.1 Overview 6
   - 5.2 Architecturally Significant Design Packages 7

6. Interface Description 9

7. Size and Performance 9

8. Quality 9

Software Architecture Document

1. Introduction

*[The introduction of the **Software Architecture Document** provides an overview of the entire **Software Architecture Document**. It includes the purpose, scope, definitions, acronyms, abbreviations, references, and overview of the **Software Architecture Document**.]*

The Software Architecture Document provides a comprehensive architectural overview of the Arithmetic Expression Evaluator system. The purpose of this document is to capture the significant architectural decisions made for the system and convey them through multiple architectural views. The scope of this document covers the full architecture of the Arithmetic Expression Evaluator, including components for expression parsing, evaluating expressions, handling operators and precedence, managing errors, and the user interface. Definitions of terms and acronyms used in the document, such as PEMDAS (Order of Operations), are included to assist in properly interpreting the information provided. References are provided to related documents, such as the Arithmetic Expression Evaluator Project Description, to give additional context. An overview of the contents of the Software Architecture Document is provided, summarizing the key sections: Architectural Representation, Architectural Goals and Constraints, Logical View, Interface Description, Size and Performance, and Quality. The remainder of the document will provide the details for each of these architectural views and perspectives of the system.

1.1 Purpose

[This document provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which have been made on the system.]

*[This section defines the role or purpose of the **Software Architecture Document**, in the overall project documentation, and briefly describes the structure of the document. The specific audiences for the document are identified, with an indication of how they are expected to use the document.]*

This document provides a comprehensive architectural overview of the Arithmetic Expression Evaluator system. It captures the significant architectural decisions made for the system and conveys them through multiple architectural views. The intended audience is the project team, who will use it to understand the overall software architecture and ensure implementation aligns with the design.

1.2 Scope

[A brief description of what the Software Architecture Document applies to; what is affected or influenced by this document.]

This document covers the architecture for the full Arithmetic Expression Evaluator system. This includes components for expression parsing, evaluating expressions, handling operators and parentheses, managing errors, and the user interface.

1.3 Definitions, Acronyms, and Abbreviations

SRS – Software Requirements Specification

PEMDAS – Mathematical order of operations (Parentheses, Exponents, Multiplication/Division, Addition/Subtraction)

EECS – Electrical Engineering and Computer Science (as related to the department at KU)

GCC – GNU Compiler Collection (as related to C programs)

G++ – Another compiler; Like GCC but for C++

1.4 References

*[This subsection provides a complete list of all documents referenced elsewhere in the **Software Architecture Document**. Identify each document by title, report number (if applicable), date, and publishing organization. Specify the sources from which the references can be obtained. This information may be provided by reference to an appendix or to another document.]*

1. https://cplusplus.com/reference/stack/ – C++ Stack header

2. EECS 348 Project Description, Version 1.0, Professor Hossein Saiedian, Department of Electrica Engineering and Computer Science, University of Kansas, Lawrence, KS, Fall Semester 2023.

1.5 Overview

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

This document covers our software design architecture. It will touch on various items such as size, interface, and actual structure. The Software Architecture Document contains the following sections:

Architectural Representation: Describes the architectural views and models used to represent the system .

Architectural Goals and Constraints: Captures architectural requirements and objectives

Logical View: Provides an overview of the system's decomposition into packages and significant classes

Interface Description: High-level details of key interfaces

Size and Performance: Dimensions and performance factors relevant to architecture

Quality: How the architecture supports quality requirements such as extensibility and reliability

2. Architectural Representation

[This section describes what software architecture is for the current system, and how it is represented. It enumerates the views that are necessary, and for each view, explains what types of model elements it contains.]

The architecture for the system is object-oriented architecture. The reason for this decision is due to the way the software is designed. The division of responsibility for the program reflects the object-oriented architecture style, as it is broken down into individual sections that are able to be reused.

3. Architectural Goals and Constraints

[This section describes the software requirements and objectives that have some significant impact on the architecture; for example, safety, security, privacy, use of an off-the-shelf product, portability, distribution, and reuse. It also captures the special constraints that may apply: design and implementation strategy, development tools, team structure, schedule, legacy code, and so on.]

- Safety: Regarding safety, the requirement is to ensure that the code runs without error and making sure that any error in the input is caught and dealt with correctly to guarantee that nothing impacts the computer machine that the program is run on.
- Distribution: Regarding distribution, the requirement is to ensure that the code is easily distributed to all needed recipients and to ensure that the distribution method is easy to use and easy to replicate.
- Reuse: Regarding reuse, the requirement is that the program is designed in a way such that various parts may be reused for future programs. This ensures that the programming itself can be done efficiently and future programs can have a foundation that helps the focus be on the specifics of the task and not on the mundane tasks of formatting and various things of that nature.
- Design and Implementation: As mentioned in the above section, the main design and implementation will be built around the object-oriented architecture. This system for the design and implementation was selected due to the breakdown of the sections and way to code the program, and the fact that the program can be easily reused.
- Development tools: The development tools that will be used include Visual Studio Code and Github.

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 describes the architecturally significant parts of the design model, such as its decomposition into subsystems and packages. And for each significant package, its decomposition into classes and class utilities. You should introduce architecturally significant classes and describe their responsibilities, as well as a few very important relationships, operations, and attributes.]

5.1 Overview

[This subsection describes the overall decomposition of the design model in terms of its package hierarchy and layers.]

The object-oriented design of the program includes subsystems divided out to perform specific functions which provide the program's required functionalities. These subsystems include Expression Parsing, Expression Evaluation, Operator Handling, Error Management, and User Interface.

The User Interface provides the user access to the program. They can input their desired calculations, which are then passed to the Expression Parser. This package determines this input as either valid, in which case it is sent to Operator Handling, or invalid, in which case it's sent to Error Management. Once the operators have been organized, the program passes the expression to Expression Evalution. The results are passed back to User Interface, which determines how to provide the user with their formatted output.

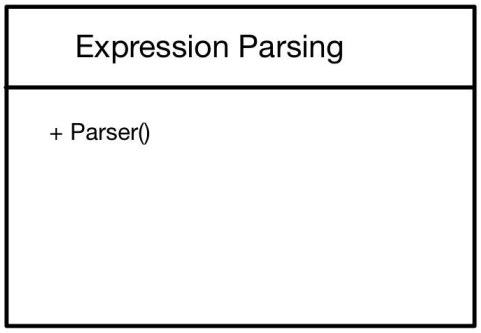
5.2 Architecturally Significant Design Modules or Packages

[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.]

Expression Parsing

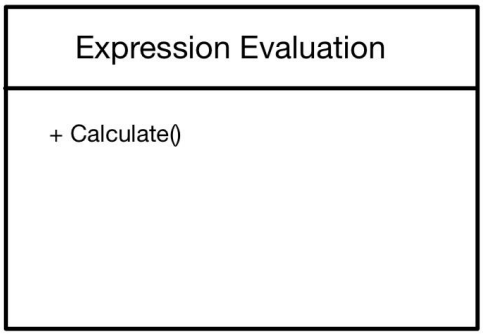
This package parses arithmetic expressions.



The Parser() class takes in user input and validates it via package connections to determine whether or not it can be sent to the Operator Handler.

Expression Evaluation

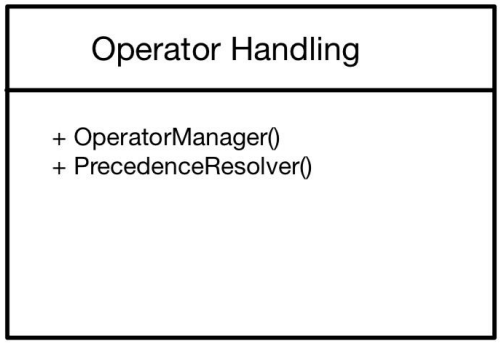
This package evaluates parsed expressions received from Expression Parsing after successfully returning from Operator Management.



The Calculate class will use the input provided to solve arithmetic operations.

Operator Management

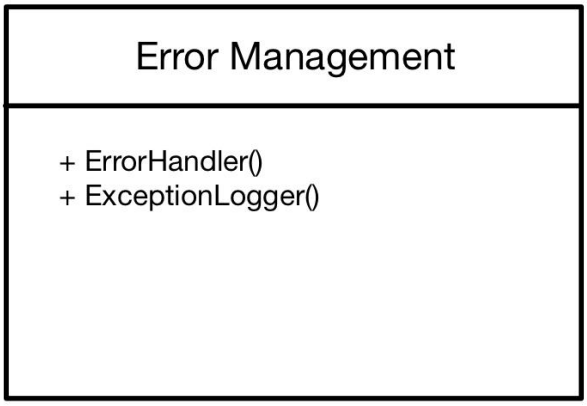
This package handles the logic of input operations to ensure solutions are found in the order of PEMDAS.



The OperatorManager class will identify and declare to the program an input's operators. The PrecedenceResolver class will use identified operators from an input expression and provide valid formatting for the program to solve the expression in the right order.

Error Management

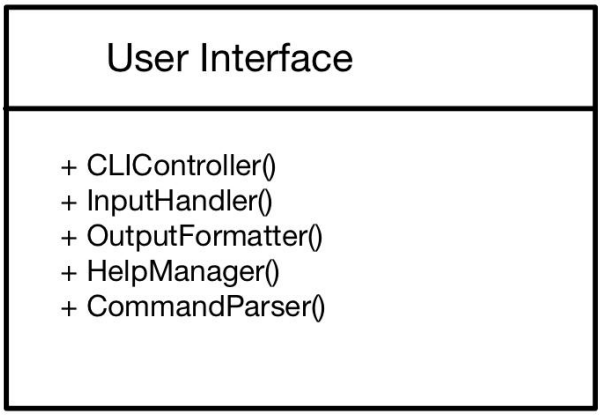
This package manages errors during parsing and evaluation.



The ErrorHandler class will instruct the program how to behave depending on what error is identified in the input. The ExceptionLogger class will report any exceptions reached in runtime.

User Interface

This package provides the interface for user interaction.



The CLIController class will manage the command line interaction for the user. The InputHandler class will direct input to the Expression Parsing and Error Management packages. The OutputFormatter class will receive input from packages and determine how to format the output intended for the user. The HelpManager class will provide assistance for the user in case command line usage of the program is unclear. The CommandParser class will prepare user input before it is sent to Expression Parsing and Error Management.

6. 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 primary user interface will be a simple prompt/command line interface. You'll run the program, and you'll be presented with this prompt where you can type your expression. Once the program evaluates the expression and prints the result to a new line, the prompt will return on another new line so the user can type the next expression.

A valid input to the prompt will be a mathematical expression which can be evaluated down to a number. In the current planned final version of the program, the output for a given input will be a number, not another expression.

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

[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 software architecture has no additional security or privacy implications. To improve the quality, the software is completely dockerized, and hence can be run as a virtualized machine, reducing the risk of unintentional security breaches. If the host system does not natively support Docker, for example RHEL, then Podman can be used to deploy the software.

Additionally, dockerizing the software improves the extensibility and portability since multiple operating systems can be supported. The software comes with a test suite covering significant edge cases which automatically runs when the software is first deployed, hence improving reliability and quality. If the in-built test suite fails, a warning is issued to the user in the console.

The software architecture will be checked meticulously for quality to make sure that the desired capabilities are well incorporated into the system.