EvalEx

Software Architecture Document

Version 1.0

Revision History

| **Date** | **Version** | **Description** | **Author** |
| --- | --- | --- | --- |
| 12/11/23 | 1.0 | Added all the initial project software architecture for EvalEx | Tommy Lam, Will Whitehead, Joshua Lee, Max Biundo, Joe Hotze, Gunther Leuchtefeld |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

1. Introduction 4

1.1 Purpose 4

1.2 Scope 4

1.3 Definitions, Acronyms, and Abbreviations 4

1.4 References 4

1.5 Overview 4

2. Architectural Representation 4

3. Architectural Goals and Constraints 4

4. Use-Case View 4

4.1 Use-Case Realizations 5

5. Logical View 5

5.1 Overview 5

5.2 Architecturally Significant Design Packages 5

6. Interface Description 5

7. Size and Performance 5

8. Quality 5

Software Architecture Document

# Introduction

## Purpose

This Software Architecture Document (SAD) offers a thorough architectural overview of EvalEx, an arithmetic expression evaluator developed in C++. It is designed to capture and convey the critical architectural decisions made during the development of EvalEx.

The primary role of this document is to serve as a key component of the overall project documentation. It provides a structured framework for the software's design and serves as a blueprint for both the development and maintenance phases. The SAD is intended to guide project stakeholders through the software's architecture, elucidating how the system's components interact and integrate to fulfill the project's objectives.

The specific audiences for this document include:

* Development Team: Software developers and engineers will use this document as a guide for understanding the system's structure, its components, and their interactions. It will serve as a reference point for implementing the system's functionalities according to the architectural plan.
* Project Manager: The project manager will utilize this document to oversee the project's progress and ensure that the development aligns with the planned architecture. It will also aid in resource allocation and risk management related to the technical aspects of the project.
* Quality Assurance Team: This team will refer to the SAD to understand the system's architecture for developing effective testing strategies. The document will help in identifying critical modules and interfaces where rigorous testing is required.
* Maintenance Engineers: Post-deployment, maintenance engineers will consult the SAD for insights into the system's architecture for effective troubleshooting, updating, and enhancing the system.
* Future Contributors: For individuals or teams who will contribute to the system in the future, this document will act as a comprehensive resource to understand EvalEx's architecture and underlying principles.

## Scope

The scope of this document encompasses the structural design, functional behavior, and developmental guidelines that are fundamental to the creation and implementation of EvalEx.

This document is intended to be used as a reference and guide by the development team, ensuring that all architectural decisions align with the project's objectives and requirements. The document outlines the architecture in a manner that supports maintainability, scalability, and performance, while also addressing constraints and requirements such as programming language (C++), platform specifications, and design considerations.

Overall, the scope of this Software Architecture Document is to provide a clear, structured, and detailed architectural framework for the EvalEx program, guiding the team through the complexities of software development while maintaining a focus on quality and adherence to the outlined requirements and goals.

## Definitions, Acronyms, and Abbreviations

* C++: A high-level programming language with object-oriented and generic programming features, used for developing the EvalEx program.
* PEMDAS: An acronym for Parentheses, Exponents, Multiplication and Division (from left to right), Addition and Subtraction (from left to right). This represents the order of operations used in arithmetic expressions.
* UI: User Interface. Refers to the part of the program that handles interaction with the user, such as inputting expressions and displaying results.
* CLI: Command Line Interface. A type of user interface operated entirely through a console or terminal window, using text-based commands.
* OOP: Object-Oriented Programming. A programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as attributes), and code, in the form of procedures (often known as methods).
* API: Application Programming Interface. A set of routines, protocols, and tools for building software applications, specifying how software components should interact.
* SDLC: Software Development Life Cycle. A process for planning, creating, testing, and deploying an information system.
* UML: Unified Modeling Language. A standardized modeling language enabling developers to specify, visualize, construct, and document artifacts of software systems.
* IDE: Integrated Development Environment. A software suite that consolidates basic tools required for software testing and development.
* VCS: Version Control System. A system that records changes to a file or set of files over time so that specific versions can be recalled later.
* GIT: A distributed version-control system for tracking changes in source code during software development.
* Unit Testing: A level of software testing where individual units/components of a software are tested independently.

## References

Team Happiness Club Software Development Plan - **[1]**

Team Happiness Club Software Requirements Specification - **[2]**

## Overview

The rest of the document is organized into several key sections, each designed to offer a detailed perspective on different aspects of EvalEx's architecture:

* 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.
* Architectural Goals and Constraints: This section describes the software requirements and objectives that have some significant impact on the architecture. It also captures the special constraints that may apply: design and implementation strategy, development tools, team structure, schedule, legacy code, and so on.
* 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. It introduces architecturally significant classes and describes their responsibilities, as well as a few very important relationships, operations, and attributes.
* Interface Description: A description of the major entity interfaces, including screen formats, valid inputs, and resulting outputs.
* 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.

# Architectural Representation

Logical View

* Purpose: Focuses on the functionality that the system provides to end-users.
* Model Elements:
  + Classes: Tokenizer, ExpressionTreeBuilder, ExpressionTree, Operator, AdditionOperator, SubtractionOperator, MultiplicationOperator, DivisionOperator, ModuloOperator, ExponentiationOperator, Evaluator, CommandLineInterface, ErrorHandling.
  + Packages: Main Package (EvalEx), Expression Parsing (EvalEx.Parsing), Operator Handling (EvalEx.Operators), Expression Evaluation (EvalEx.Evaluation), Error Handling (EvalEx.Errors).
  + Class Relationships: There is a inheritance relationship between the parent Operator class and the children classes AdditionOperator, SubtractionOperator, MultiplicationOperator, DivisionOperator, ModuloOperator, and ExponentiationOperator,

Process View

* Purpose: Addresses the system's concurrency and synchronization aspects.
* Model Elements:
  + Threads and Processes: There will be no parallel processing since this program is not complex enough to garner this.

Development View

* Purpose: Focuses on the system's software modules and subsystems.
* Model Elements:
  + Software Modules: Tokenizer, Parser, Evaluator, ErrorHandler.

# Architectural Goals and Constraints

Our goal is to create a program that can calculate complex mathematical equations. The program should be able to add, subtract, multiply, divide, and correctly use exponents. The program should also be able to follow PEMDAS, also known as order of operations, to come to the correct conclusion while calculating equations with more than one operator. The program won’t be capable of handling algebraic expressions. The program cannot do integrals or derivatives. The program should only take data from user input. No other user data should be collected such as IP address, usernames and passwords, or any other data. Any sessions will not be shared or collected for any use. The user’s data is completely private from the public. With no user data stored, there is no need to invest in security, as there is no information at risk. The program should be able to be used on any computer that runs C. Mobile devices will be excluded from this. This program exists for the sole purpose of the project, and therefore will not be distributed to anyone except those in charge of grading the assignment. The program can be reused in the future in order to build a different project if needed. This program must be written using C. The program also must be completed in slightly less than one month’s time in order to reach the deadline. Our schedule will be tight over the next four weeks. We will develop each operator and function. Once that is done, we will develop the interface, and finish the project. Outside VS Code, no other development tools will be necessary. No legacy code is relevant for this project.

# Logical View

## Overview

The design model of EvalEx, our Arithmetic Expression Evaluator, is structured in a hierarchical and layered manner, ensuring clarity, maintainability, and separation of concerns. This section describes the overall decomposition of the design model in terms of its package hierarchy and layers.

Package Hierarchy

* Main Package (EvalEx): The root package that encompasses all other packages and serves as the entry point for the application.
* Expression Parsing (EvalEx.Parsing): This package is responsible for parsing input expressions. It includes classes and interfaces for tokenization and syntax analysis. Key components include:
  + Tokenizer: Breaks down the input string into tokens (numbers, operators, parentheses).
  + Parser: Constructs an abstract syntax tree (AST) from tokens.
* Operator Handling (EvalEx.Operators): This package contains the logic for handling different arithmetic operations. Each operator (+, -, \*, /, %, ^) is encapsulated in its class, inheriting from a common interface.
  + AdditionOperator, SubtractionOperator, etc.: Classes for each arithmetic operator.
* Expression Evaluation (EvalEx.Evaluation): This package deals with evaluating the parsed expressions. It traverses the AST and computes the result.
  + Evaluator: Evaluates the expression based on the AST generated by the Parser.
* Error Handling (EvalEx.Errors): Dedicated to managing exceptions and errors, such as syntax errors or division by zero.
  + ErrorHandler: Catches and processes exceptions, providing meaningful error messages.

Layers

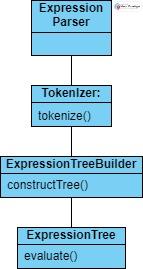
* User Interface Layer: The topmost layer that interacts with the user. It takes input expressions and displays results or error messages. This layer communicates with the Parsing layer to initiate the expression evaluation process.
* Parsing Layer: Parses the input expressions. It includes tokenization and syntax analysis, forming an AST. This layer passes the AST to the Evaluation layer for computation.
* Evaluation Layer: Responsible for traversing the AST and computing the result of the expression. This layer utilizes the Operator Handling layer to deal with specific arithmetic operations.
* Operator Handling Layer: Encapsulates the logic for each arithmetic operation. It is utilized by the Evaluation layer to perform specific calculations as per the parsed expression.
* Error Handling Layer: Catches and manages errors throughout the application. It ensures that any exceptions in parsing or evaluation are handled gracefully, providing informative feedback to the user.

This hierarchical and layered approach in EvalEx ensures that each component has a specific role, contributing to a clean, manageable, and extendable architecture. The separation of concerns is evident, with each package and layer focusing on its designated functionality, which aids in both development and maintenance phases.

## Architecturally Significant Design Modules or Packages

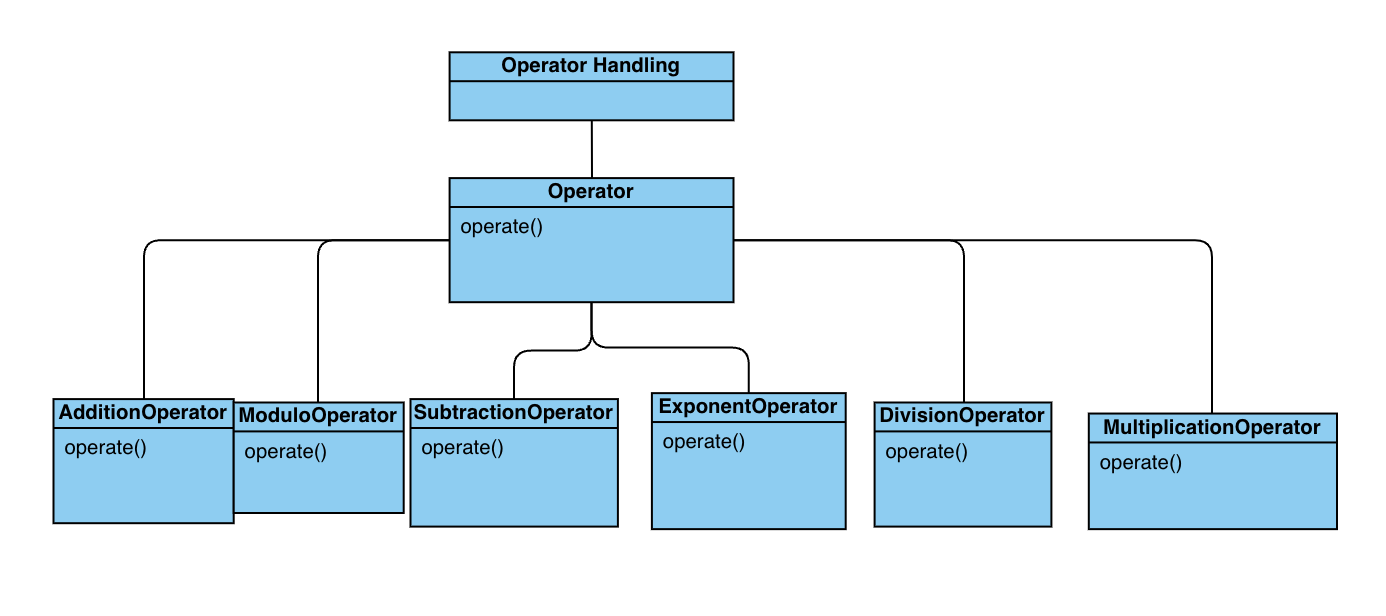
1. Expression Parser

* Description: This package is responsible for parsing arithmetic expressions input by the user. It tokenizes the input and builds a structured representation of the expression for evaluation.
* Classes:
  + Tokenizer:
    - Description: Splits the input expression into tokens (operators, parentheses, and numbers).
    - Major Responsibilities: Identifying and categorizing tokens, handling input errors.
  + ExpressionTreeBuilder:
    - Description: Constructs an expression tree from the tokenized input.
    - Major Responsibilities: Implementing the shunting-yard algorithm to convert infix expressions to postfix and building the expression tree.
  + ExpressionTree:
    - Description: Represents the structured form of the parsed expression.
    - Attributes: Root of the tree.
    - Operations: Evaluate the expression tree, print the tree structure.

***Expression Parser Diagram***

2. Operator Handling

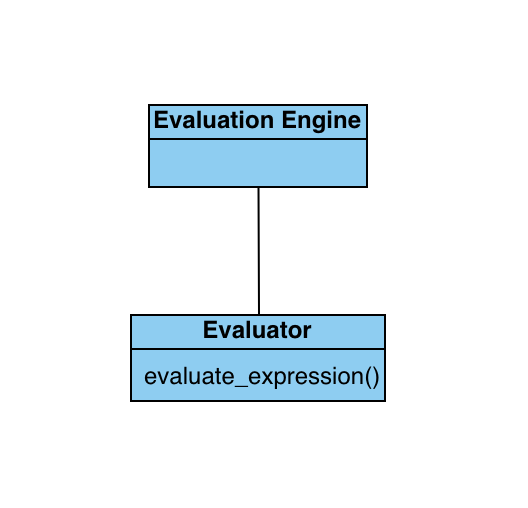
* Description: Manages the implementation and precedence of arithmetic operators.
* Classes:
  + Operator:
    - Description: Abstract base class for different arithmetic operators.
    - Major Responsibilities: Define a common interface for all operators.
  + AdditionOperator, SubtractionOperator, MultiplicationOperator, DivisionOperator, ModuloOperator, ExponentOperator:
    - Description: Inherits from Operator; implements specific arithmetic operations.
    - Operations: Calculation method for the specific operation.



***Operator Handling Diagram***

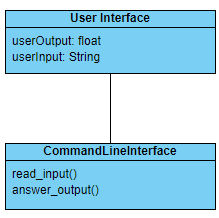
3. Evaluation Engine

* Description: Responsible for evaluating parsed expressions.
* Classes:
  + Evaluator:
    - Description: Drives the evaluation process of the expression tree.
    - Operations: Evaluate expressions, handle errors like division by zero.

***Evaluation Engine Diagram***

4. User Interface

* Description: Manages user interactions and displays results.
* Classes:
  + CommandLineInterface:
    - Description: Provides a command-line interface for user input and output.
    - Operations: Read user input, display results, error messages.

***User Interface Diagram***

# Interface Description

The interface will be a basic terminal design. There will be no extra buttons or interfaces for the program. Every input will be typed into the terminal by the user. All numbers, and the +, -, \*, /, and ^ will be accepted as good inputs. The program should output the number with no extra information. In the case of invalid input, a message should be displayed saying what the problem with the code is. This message should be as concise as possible, but still make clear what is wrong with the user’s operation. This will cover errors such as dividing by zero or any other misuse of operators or bad input.

# Quality

* Extensibility:
  + Extensibility refers to the ease with which the system can be extended or enhanced.
  + The program will contain all of the necessary operators specified within the instructions, but should more be added, it will be a simple matter of creating additional functions that can operate on given values provided through the calculator.
* Reliability:
  + The program will be extensively tested in order to insure reliability for users. Error messages will be output in the command line interface, detailing what syntactical error the user has made in order and providing the original text so the user can fix their mistake.
* Portability:
  + The program is constructed in C++, and is therefore usable on any platform that can run it as such. The program will be available on GitHub and created as an executable, so anyone can easily access it should they be given the location to find and download it.
* Safety, Security, and Privacy:
  + Safety, security and privacy is paramount for any modern program that hits the market. That being said, a calculator program is very hard to misuse to the point of system damage. The bare minimum of preventing memory leaks should be sufficient in order for EvalEx to meet the standards of modern programs.