The University of Kansas

Arithmetic Expression Evaluator
Software Architecture Document

Version <2.0>

Revision History

Date	Version	Description	Author
06/11/23	1.0	Initial version of the Software Architecture Document	Hannah Smith
12/2/23	2.0	Updated Software architecture Document	Magaly Camacho & Christina Sorenson

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

This document aims to establish a shared understanding of the system's architectural design, serving as a reference for programmers throughout the project's progression. It covers architectural views, project goals and constraints, logical system review, package breakdown, user interface, and product quality.

1.2 Scope

This document provides an architectural overview of the Arithmetic Expression Evaluator. It affects the overall design of the project and is a good resource to identify the general structure of the final product.

1.3 Definitions, Acronyms, and Abbreviations

Software Architecture – refers to the structure and design of a software system and involves decisions about the organization of components and how they interact; provides a blueprint for building and maintaining a software application, ensuring that it meets its functional and non-functional requirements

Logical View – representation of a software system's structure and organization from a high-level, abstract perspective; provides a conceptual understanding of how a software system works

Package – mechanism for organizing and grouping related code and resources together as well as managing the complexity of large software systems by providing a hierarchical structure for organizing files and classes

Subsystems - self-contained, modular components or smaller systems within a larger, more complex system

User Interface – point of interaction between a user and the software application

Constraints – limitations, rules, or restrictions that must be adhered to during the design, development, or operation of a system

Interface – programming construct or contract that defines a set of methods, properties, and/or events that a class or object must implement; defines a common set of functionalities that multiple classes can adhere to

Object-oriented Programming (OOP) – programming paradigm or methodology that uses objects, which are instances of classes, to design and structure computer programs

Tokenization – process of breaking down a sequence of text into individual units, known as tokens; divides text into smaller, meaningful units to facilitate the analysis

1.4 References

Project Management Plan, 09/22/2023, University of Kansas

This Software Development Plan serves as a document aimed at collecting the essential details needed for the Arithmetic Expression Evaluator project's control and management. It outlines the strategy being used for software development in this project and stands as the primary plan that team members rely on to guide the project's progress.

Software Requirements Specifications, 10/09/2023, University of Kansas

The Software Requirements Specifications document is a document that describes what and how the software will perform. It also takes stakeholders' needs into account and describes the functionality needed to fulfill these needs. Finally, the Software Requirements Specifications document outlines nonfunctional requirements, design constraints, and anything else needed to give a complete description of the software requirements.

1.5 Overview

The rest of the Software Architecture Document contains the following sections in the given order: Architectural Representation, Architectural Goals and Constraints, Logical View, Interface Description, and Quality. For this Arithmetic Expression Evaluator, neither Use-Case View nor Size and Performance will be discussed.

2. Architectural Representation

There is one main way to represent the architecture for the Arithmetic Expression Evaluator. This is the logical view – this is the manner in which the architecture is decomposed into its individual parts. There are two primary packages; one is end-user facing and the other is back-end facing (containing one primary class).

3. Architectural Goals and Constraints

The software requirements that significantly impact the architecture include expression parsing, operator support, operator precedence, parenthesis handling, numeric constants, and the user interface. The main characteristics of the goals and constraints for the architecture are expression handling, tokenization, and calculation.

4. Use-Case View

N/A

4.1 Use-Case Realizations

N/A

5. Logical View

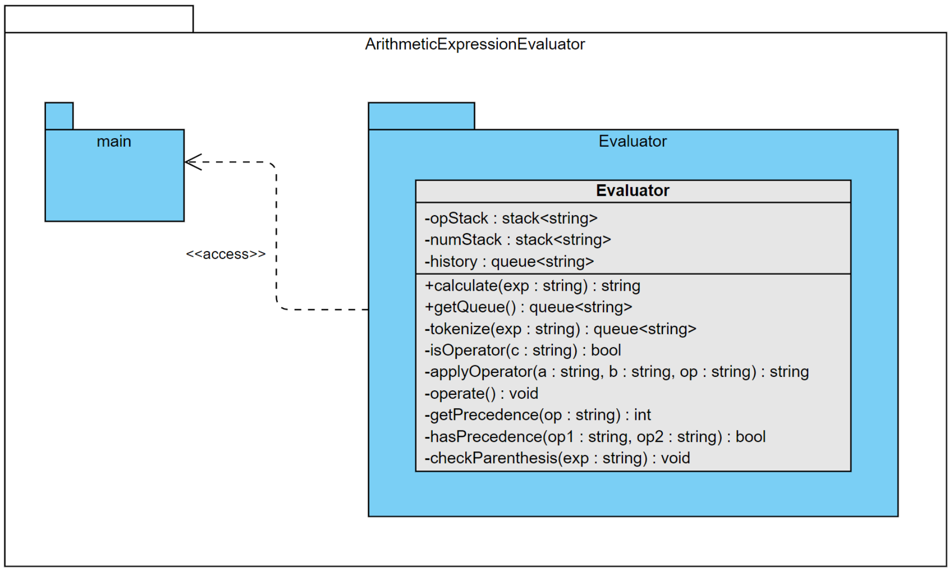
5.1 Overview

This architecture design has a single package that then contains two other packages. One package is focused on interfacing with the user, taking an input and returning the correct response. The other package is focused on tokenizing the expression and then evaluating it.

5.2 Architecturally Significant Design Modules or Packages

5.2.1 Arithmetic Evaluator

This is the main package which will contain the Main (User Interface) package and the Evaluator package.


ArithmeticExpressionEvaluator
main
<<access>>
Evaluator
Evaluator
-opStack : stack<string>
-numStack : stack<string>
-history : queue<string>
+calculate(exp : string) : string
+getQueue() : queue<string>
-tokenize(exp : string) : queue<string>
-isOperator(c : string) : bool
-applyOperator(a : string, b : string, op : string) : string
-operate() : void
-getPrecedence(op : string) : int
-hasPrecedence(op1 : string, op2 : string) : bool
-checkParenthesis(exp : string) : void

5.2.2 Main (User Interface)

This package is designed to manage user interactions. Initially, it will provide users with general instructions regarding accepted operands. Subsequently, it will receive the user-input expression and then it will use the Evaluator package to calculate a result. In the case of an error in the expression, it will display errors to the user using easily understandable terminology.

The main file comprises:

- A main function – this function prints instructions for the user, captures their input, and then initializes an evaluator object. This process continues in a loop until the user enters "exit." The function also encompasses error detection and handling.
- printQueue - a helper function for the history function that prints the queue provided by the evaluator object.

5.2.3 Evaluator

This package includes the class Evaluator, which can be used to initialize an object of Evaluator type. This object will keep track of the successfully evaluated expressions and their results, which can be retrieved using the public method *getQueue*. The object can then be used to calculate expressions through its public method *calculate*. This method first resets the helper stacks, *opStack* and *numStack*, to empty stacks, and then it checks that all parentheses in the expression are matched, if they are not, it throws an error. The method then tokenizes the expression (while checking for invalid characters and throwing an error if so) and it iterates through the tokens while using separate stacks to keep track of the operators and the operands. The method then proceeds depending on whether the token is a number, an operator, an opening parenthesis, or a closing parenthesis: numbers are pushed to *numStack*; for operators, while the operator does not have precedence over the top of *opStack*, it calls the method *operate*; opening parenthesis are pushed to *opStack*; for closing parenthesis, it calls *operate* until an opening parenthesis is popped. After iterating through the tokens, it then calls *operate* for any remaining operators in opStack, or it raises an error if there's not enough operands. Finally, if only one number is in numStack, it returns that number as the answer, otherwise it throws an error.

5.2.3.1 The Evaluator class also has the following private attributes:

- opStack – stack used to store operators while evaluating an expression
- numStack – stack used to store operands while evaluating an expression
- history – queue used to track evaluated expressions and their results

5.2.3.2 The Evaluator class also has the following private methods:

- tokenize – takes an expression and separates it into tokens
- isOperator – checks if a given character is an operator
- applyOperator – returns result of applying a given operator on two given operands
- operate – applies the top operator in *opStack* to the two top operands in *numStack*
- getPrecedence – returns the precedence of the given operator
- hasPrecedence – returns if the first given operator has equal or greater precedence compared to the second given operator
- checkParenthesis – checks that the parentheses in the given expression are matched

6. Interface Description

The user interface operates through the command line, prompting users to input an expression for evaluation. The output displays the result beneath the input expression. Moreover, the evaluator adeptly handles errors by providing informative text to guide users in resolving their issues. Various examples are illustrated below.

6.1 Valid Expressions include:

6.1.1 Addition

Enter an Expression: 3+4
Result: 7

6.1.2 Subtraction

Enter an Expression: 8-5
Result: 3

6.1.3 Multiplication

Enter an Expression: 4*3
Result: 12

6.1.4 Division

Enter an Expression: 10/5

Result: 2

6.1.5 *Modding*

Enter an Expression: 12%5
Result: 2

6.1.6 *Exponents*

Enter an expression: 2^3
Result: 8

6.2 Invalid Expressions Include

6.2.1 *Mismatched Parentheses*

Enter an expression: (3+2
Result: Mismatched Parentheses

6.2.2 *Missing Operator*

Enter an expression: (3+4)(4+5)
Result: Missing Operator

6.2.3 *Operator Without Operands*

Enter an expression: (*3)+(/3)
Result: Operator Without Operands

6.2.4 *Invalid Character*

Enter an expression: (3#2)+5
Result: Invalid Character

6.2.5 *Incorrect Operator Usage:*

Enter an expression: 10/0
Result: Incorrect Operator Usage

6.3 Other Options are

6.3.1 *History*

Enter an Expression: History
History:
3+4 = 7
8-5 = 3
4*3 = 12
10/5 = 2
12/5 = 2
2^3 = 8

6.3.2 *Exit*

Enter an Expression: Exit

7. Size and Performance

N/A

8. Quality

All defect metrics will be gathered and recorded as Change Requests.

All deliverables will be reviewed as needed to ensure that each is of acceptable quality.

All defects found after release/integration will be recorded and tracked as post-integration Change Requests.

Security will be managed via public and private methods among classes.

The architecture is structured such that additional operators can be implemented to improve the quality of the final product.