EECS 348 Group 16

# Arithmetic Expression Calculator Software Architecture Document

Version 1.2

# Revision History

| Date | Version | Description | Author |
|---|---|---|---|
| 12/11/2023 | 1.0 | Initial document creation. | Sam Muehlebach, Josh Welicky, Jennifer Aber, Mark Kitchin, Jawad Ahsan, Basim Arshad |
| 28/11/2023 | 1.1 | Added the terms infix and postfix to section 1.3. | Josh Welicky |
| 29/11/2023 | 1.2 | Updated the package diagram and descriptions of the Stack, Vector, and Driver classes | Sam Muehlebach, Josh Welicky |
|  |  |  |  |

# Table of Contents

# Software Architecture Document

## 1. Introduction

### 1.1 Purpose

This document provides an overview of the software architecture of the Arithmetic Expression Calculator, depicting the system through the logical view. This document outlines the decisions made in regard to the software's design as well as the reasoning for those decisions.

### 1.2 Scope

This document applies to the structure and functionality of the code, particularly the files, classes, attributes, and methods that are involved in evaluating the input operations. It affects the composition of the software code when completing the project objective as well as the decomposition into separate packages, and is influenced by the previous and accumulating strategies, constraints, and goals of the project.

### 1.3 Definitions, Acronyms, and Abbreviations

| | | |
|---|---|---|
| Class | — | A data object that represents an abstract concept. A class contains attributes and methods(operations). A class can also make some elements public(accessible) while keeping other private. |
| Class Diagram | — | A kind of diagram that outlines the names, attributes, and methods of a class, as well as the relationships between multiple classes. |
| Infix Notation | — | A mathematical notation where an operator is placed in between both operands. For example, 3+4 is in infix notation. |
| Object Oriented Programming | — | An architectural style that decomposes the software into several  runtime components, such as classes. |
| Package | — | A grouping of code elements, in this case, those that are isolated to a separate file or collection of files. |
| Package Diagram | — | A kind of diagram that outlines the relationships between different packages of a software, as well as the elements in such packages. |
| Postfix Notation | — | A mathematical notation where an operator is placed after its operands. For example, 3 4 + is the postfix version of 3 + 4 |
| UML | — | Short for Unified Modeling Language. A design language used to create Package and Class diagrams. |

### 1.4 References

| | | |
|---|---|---|
| Project Description | — | Can be found in the EECS 348 Canvas page in the project file. |

### 1.5 Overview

| | | |
|---|---|---|
| Architectural Representation | — | Describes the software architecture and its representation. Describes the views of the architecture and the models used to represent the architectural design. |
| Architectural Goals and Constraints | — | Describes the factors that have had a significant impact on the software architecture. |
| Use-Case View | — | Not applicable. |
| Logical View | — | Describes the significant components, packages, and classes of the software architecture. |
| Interface Description | — | Describes the user interface, input and output formats. |
| Size and Performance | — | Not applicable. |
| Quality | — | Description of how the software architecture contributes to factors other than functionality. |

## 2. Architectural Representation

The software architecture presented in this document is utilized for the initial implementation of the Arithmetic Expression Calculator. This document does not utilize a use-case view to represent this architecture. The logical view of the software architecture is presented in section five. In this view, the software architecture is decomposed into separate packages, each of which contains a class that performs a subset of functionality. This decomposition is presented using a UML package diagram, which also contains the elements of a UML class diagram to show the relationships between the different classes in the different packages.

## 3. Architectural Goals and Constraints

Goals- Code and architecture successfully meets the requirements of the project, as specified in the Project Description document, such as correctly evaluating operations, handling errors caused by invalid input, obtaining input, and correctly displaying the result of operations.

Constraints- The architecture must utilize object-oriented programming principles, C++ programming, and the means to obtain user input. The design must be simple enough to allow ample time for implementation and testing to be completed by December 3rd, 2023. In addition, the software must be designed in a way so that all team members have a share of the implementation effort. The design must also be created in a way so that all software components can be stored on and retrieved from a GitHub repository.

## 4. Use-Case View

### 4.1 Use-Case Realizations

Not applicable

## 5. Logical View

### 5.1 Overview

The design model is decomposed into four packages. The driver package contains the Driver class as well as the main() function. This class will be responsible for driving the program by obtaining user input, parsing the input, calculating and displaying the result of the input expression. The arithmetic package contains the Arithmetic class. It currently contains the functionality for the required arithmetic operations. The vector package contains the Vector class, which is used to store an input arithmetic expression. The stack package contains the Stack class, a subclass of Vector that is used to divide the entered expression into subexpressions that can be evaluated in the proper order of operations.

### 5.2 Architecturally Significant Design Modules or Packages

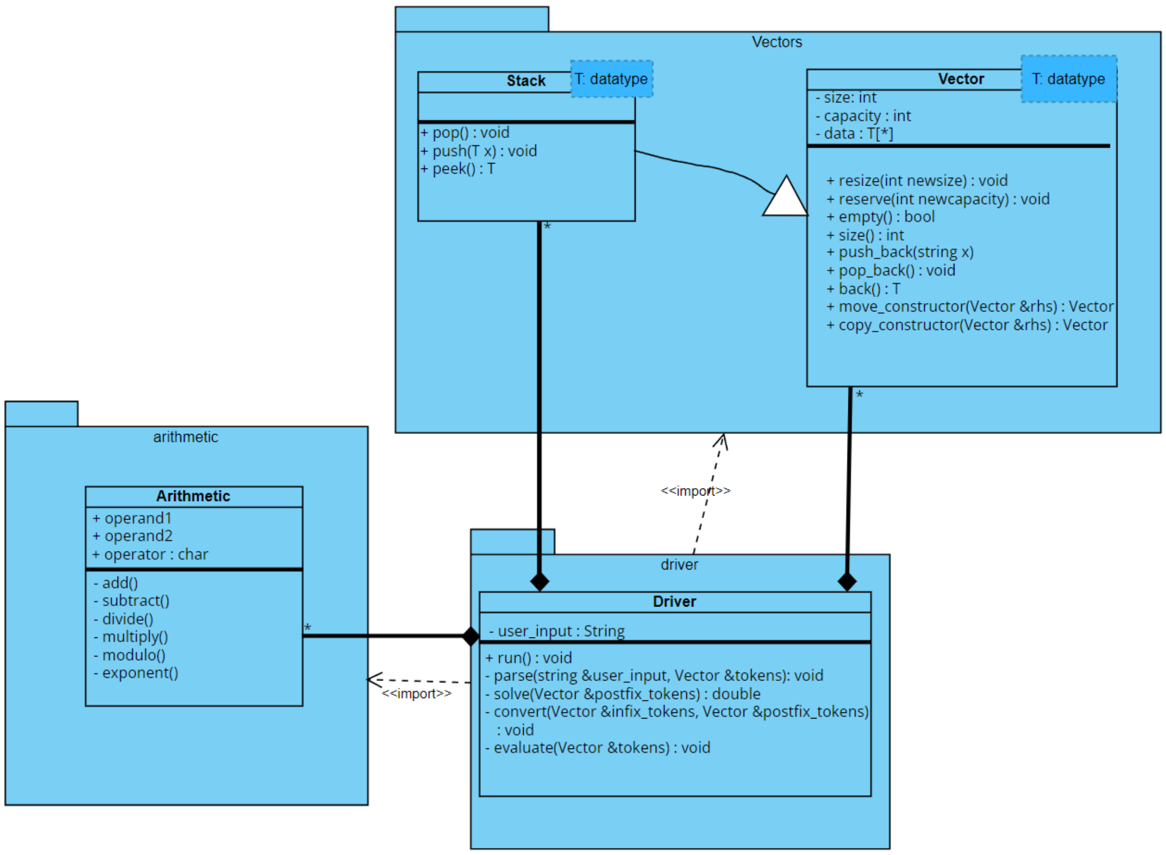

Figure 1: UML Package Diagram of the project.

**Vector** – The Vectorspackage contains the Vector class. The Vector class allows the user to enter mathematical expressions of varying lengths. The arithmetic calculator accepts an expression as string input, which it converts to a vector of tokens. Vectors share similarities with arrays in that they allow elements to be accessed individually by assigning an index value to each element. However, if input is stored in an array, the size of the array will be fixed, which could cause overflow errors at runtime. One option would be to set the size of the array such that it would be unlikely that the user would enter an expression that exceeds the allotted space, but the Vector class, which contains the resize() function for dynamic memory storage, is a far more efficient option.

**Attributes of the Vector class**:

- *Size*: The number of elements contained in the vector. In this case, the elements would be components of the expression – integers 0-9, +, -, /, *, ), (, and so forth.
- *Capacity*: The number of elements the vector can hold.
- *Data*: The array of elements contained in the vector, which could be stored as characters, integers, or floating point numbers.

**Significant Methods of the Vector Class**:

- *void resize(int newSize)*: Calls the *reserve()* function. Sets the *Size* attribute to *newSize.*
- *void reserve(int newCapacity):* Creates an empty vector of size *newCapacity*, and calls the move constructor to copy the data from the old vector to the new, temporary vector. The *Capacity* attribute becomes *newCapacity*, and the data is copied over from the temporary vector to the original, now resized, vector.
- *bool empty()*: Returns true if *Size* == 0.
- *int size()*: Returns the value of *Size*.
- *int capacity()*: Returns the value of *Capacity*.
- *void push_back(const Datatype& x):* Inserts a data element at the end of the vector. Calls *resize()* as needed.
- *void pop_back()*: Removes the last element in the vector. Decreases *Size* by 1.
- *Datatype& back()*: Returns the last element in the vector.
- *MyVector(const MyVector& rhs):* Copy constructor, used when the "copy" keyword is used. It creates an exact duplicate of a vector. Note: rhs is the original vector we are making a copy of.
- *MyVector(MyVector&& rhs):* Move constructor used when the "std::move" keyword is used. It creates a duplicate of a vector, then deletes the original vector. The rhs parameter is the vector being moved.

**Stack** – The Vectors package also includes the Stack class and imports the vector package. The Stack class is a subclass of the Vector class in that it can contain several items, and each item can be added to the stack separately and in a particular order. However, whereas items in a vector can be accessed by position number, items in a stack are accessed by the order in which they were added to the stack. Items are removed from a stack in the LIFO (Last In First Out) ordering, where the most recently added item is removed first. This is useful for operations where a mathematical expression is evaluated using postfix notation, because items can be removed from the stack in a specific sequence and fed into functions that perform arithmetic operations. The Stack does not contain any attributes, but it is implemented as a subclass of the Vector class, so attributes of the Vector class would apply.

**Significant Methods of the Stack Class**:

- *void pop():* Removes the top item in the stack.
- *void push(Datatype& x)*: Adds a new item to the top of the stack.
- *Datatype peek()*: Returns the top item in the stack without removing it.

**Arithmetic-** The arithmetic package contains the Arithmetic class. The Arithmetic class will perform all arithmetic operations. It will take in two operands and an operator. It will analyze the entered operator and immediately perform and return the result of the operation upon creation of an instance. The functionality for each operation will be defined in a separate method and performed upon the operand attributes. The arithmetic class is also in charge of handling zero division and null operand errors. All methods will be templatized so they can accept floats when applicable.

**Significant Methods of the Arithmetic Class:**

- *operand1:* This will be the number in an expression to the left of the operator.
- *operand2:* This will be the number in an expression to the right of the operator.
- *operator*: This will be the operator of the expression, a character like "+" or "-"

**Significant Methods of Arithmetic Class:**

- *add()*: Returns the sum of the operands.
- *subtract()*: Returns the difference between the operands.
- *divide()*: Returns the division between operands. Handles the zero-division error.
- *multiply()*: Returns multiplication between operands.
- *modulo()*: Handles the modulo between operands.
- *exponent()*: Returns the result of taking the first operand to the power of the second.

As templatized methods, their return type will vary based on the type of the operands.

**Driver** – The driver package contains the Driver class. In addition, it imports the vector, stack, and arithmetic packages. The Driver class handles storing, tokenizing, and evaluating the mathematical expressions input by the user. When the end user inputs an expression, it is stored as an infix expression in the form of a string. The string is separated into characters and pushed onto a vector. It is then converted to postfix using the Stack class. To obtain the final result, the solve() method will then begin to evaluate the subexpressions stored in the stack until a final result is obtained. To evaluate these subexpressions, their individual components are passed to the Arithmetic class, which will immediately return the result. This process is directed by the run() method, which takes control of obtaining input and calling the other methods. The run() method performs most of the functionality that would otherwise be delegated to a globally declared main() function. Therefore, the main() function would only be responsible for creating an instance of Driver and calling its run() method. The driver class will create and destroy all instances of Vector, Stack, and Arithmetic.

**Significant Attributes of the Driver class:**

- *user_input:* This is the string entered by the user. It will be the initial form of the arithmetic expression to be evaluated.

**Significant Methods of the Driver class:**

- *void run()*: Directs the obtaining, parsing, and evaluation of user input.
- *void tokenize(*string& userinput, Vector<string>& tokens*)*: Checks if the input is valid and converts it into an infix vector.
- *void convert(Vector<string>& infix_tokens, Vector<string>& output_tokens)*: Converts an infix vector to a postfix vector.
- *Double evaluate(Vector<string> & tokens):* Searches the tokenized input for any errors, such as improperly placed operators, operands, and parentheses. Also checks for mismatched parentheses.

- *double solve(Vector<string>& postfix_tokens)*: Ultimately evaluates the stack-divided expression, returning the end result.

## 6. Interface Description

The main interface of this project will be a command line interface, where a user is prompted to type an arithmetic expression for the program to evaluate. The valid inputs consist of a combination of supported operands, matching parentheses, and supported operators that are followed by an operand or an internal operation. For instance, an example of valid input is the expression "((9+6)) / ((3*1) / (((2+2))) - 1)". Valid input can only contain numeric characters or characters associated with an arithmetic operation, such as the characters +, -, and more. In addition, internal operations must be enclosed in parentheses. Extraneous parentheses may be used, but all parentheses must be matched. Input with unmatched opening or closing parentheses are invalid. Input must also follow the basic rules of algebra, namely by not dividing by zero. The output will be the result of the calculated operation. The output will be in integer form unless computation results in a floating-point number. For instance, the output of the provided input will be "-60", not "-60.0". Output will be printed to the terminal.

## 7. Size and Performance

Not applicable.

## 8. Quality

The program will be able to run without any unexpected errors that may cause it to crash. To achieve this, robust error handling will be included in the implementations of the classes described above. As such, this architecture contributes to the reliability of the software. This architecture is very modular in nature, so that individual classes can be improved and changed without affecting other components. The program will be able to run on any device that can utilize a command line interface. As such, this architecture contributes to the compatibility of the software with most operating systems. This design ensures the privacy and security of a user's inputs and results by deleting all inputs and outputs upon termination of the program. User input and output are not retained after the software terminates.