Arithmetic Expression Evaluator

Version <1.2>

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 11/10/2023 | 1.0 | Update Introduction | Elizabeth Channel |
| 11/11/2023 | 1.1 | Update Architectural Goals and Constraints | Aryan Kevat |
| 11/12/2023 | 1.2 | Update Logical View | Chris Cooper |
| 11/12/2023 | 1.3 | Update Representation | Jeffrey Burns |

Table of Contents

[1. Introduction 3](#_Toc150651747)

[1.1 Purpose 3](#_Toc150651748)

[1.2 Scope 3](#_Toc150651749)

[1.3 Definitions, Acronyms, and Abbreviations 3](#_Toc150651750)

[1.4 References 4](#_Toc150651751)

[1.5 Overview 4](#_Toc150651752)

[2. Architectural Representation 4](#_Toc150651753)

[3. Architectural Goals and Constraints 4](#_Toc150651754)

[4. Use-Case View 4](#_Toc150651755)

[4.1 Use-Case Realizations 4](#_Toc150651756)

[5. Logical View 4](#_Toc150651757)

[5.1 Overview 5](#_Toc150651758)

[5.1.1 Package Diagram 5](#_Toc150651759)

[5.1.2 Structural Description 5](#_Toc150651760)

[5.2 Architecturally Significant Design Modules or Packages 5](#_Toc150651761)

[5.2.1 Equation Element Package 5](#_Toc150651762)

[5.2.2 Control Package 7](#_Toc150651763)

[6. Interface Description 7](#_Toc150651764)

[7. Size and Performance 8](#_Toc150651765)

[8. Quality 8](#_Toc150651766)

# Introduction

The following document will detail the software architecture of the *Arithmetic Expression Evaluator (AEE)*. It will be referred to throughout the project.

## Purpose

The purpose of this document is to provide an overview of the software architecture. Using a series of subtopics such as the *Architectural Representation*, this document will act as a resource for the CAVJAC group and its programmers to refer to when regarding the architecture the *AEE*. It will detail specific architectural decisions made by the group.

## Scope

This document applies to the *AEE*. It will influence any future discussion and decisions regarding the architecture of the evaluator.

## Definitions, Acronyms, and Abbreviations

AEE – Arithmetic Expression Evaluator

## References

* Software Development Plan – 9/22/2023, CAVJAC Group
* Software Requirements Document – 10/14/2023, CAVJAC Group

## Overview

The following document will detail the software architecture of the *Arithmetic Expression Evaluator (AEE)*. It will explain components such as goals and constraints of the project, a description of the interface, and performance of the software.

# Architectural Representation

The Arithmetic Expression Evaluator will utilize a list of objects for storing and evaluating values and operators entered by the user. The input data will be separated by values and operators. The parenthesis operator will be stored as one object containing all values and operators withing the opening and closing parenthesis. The parenthesis object will be evaluated as its own equation to determine the value of the parenthesis object for the overall equation.

# Architectural Goals and Constraints

Referring to our Software Requirements Specification document, our requirements include being able to parse arbitrary user input, evaluate valid mathematical operations with correct operator precedence, and notify the user of errors in their input. The architecture must allow for the user to have a seamless experience within the program itself. The user should not need to rely on external programs or manual control flow in order to operate the program. Because the program is purely text based, the architecture will be portable and not rely on any platform specific user interface libraries.  
  
In addition to fulfilling our software requirements, there are additional constraints for our software architecture. It needs to follow object-oriented principles, including a class based design. The design should not be overly complex and should allow for completion before our (tentative) period for writing documentation, November 30th. Each team member should participate in the design and implementation of our program, including writing test cases. Finally, the project must both be open source and easily buildable using a copy of our repository hosted on GitHub.

# 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.]

## 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.]

# 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.]

## Overview

### Package Diagram

A diagram of a company

Description automatically generated with medium confidence

### Structural Description

The system is divided into two main sections: the Equation Elements, which describe the structure of the user-provided equation, and the control classes, including the Driver, Parser, and Evaluator, which interface with the user and act upon the user-provided equation.

## 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.]

### Equation Element Package

The equation element package contains classes for describing an equation. The main class is the EquationElement class – all equations are described as a list of EquationElements. The package contains several subclasses which describe the various forms an equation element could take. The Control Package creates and evaluates the list of EquationElements.

#### Class Diagram

A diagram of a company

Description automatically generated

#### Equation Element Class

This is the main class of this package – all other classes are a subclass of it. It represents one token in an equation, such as an addition operator or a number. It has three methods: *getPriority*, *getSymbol*, and *getValue*.

The *getPriority* method conveys the order in which elements should be processed when evaluating the equation. This is an integer value, with greater values having greater priority than lesser.

The *getSymbol* method returns a string which represents the equation element, such as “+” for addition.

The *getValue* method returns the value of the element. This method takes in two operands: the EquationElement that precedes the element being evaluated, and the one that proceeds it. These operands are necessary to allow polymorphism, because while a NumericValue does not require the surrounding elements to determine its value, an addition operator requires both of its surrounding elements. In addition to calculating the value of each element, this method is responsible for reporting errors based on operators’ operands, such as Division’s divide by zero error.

#### Parentheses Class

This is a subclass of EquationElement which represents parentheses. In addition to implementing all of the abstract methods from EquationElement, Parentheses has a constructor which takes in a list of EquationElements. This list is stored as its contents attribute. Parentheses uses the Control package’s Evaluator’s evaluate method to evaluate the value of its contents in order to calculate its own value.

#### Unary Operator Class

This subclass of EquationElement represents a unary operator. It does not implement any of EquationElement’s abstract methods. However, it has two subclasses: AbsoluteValue and Negation, which do. These subclasses do not warrant further explanation; for more information see the Software Requirements Document.

#### Binary Operator Class

This subclass of EquationElement represents a binary operator. It does not implement any of EquationElement’s abstract methods. However, it has six subclasses, which do: Addition, Subtraction, Multiplication, Division, Modulo, and Exponentiation. These subclasses do not warrant further explanation; for more information see the Software Requirements Document.

#### Numeric Value Class

This subclass of EquationElement represents a constant integer value. In addition to implementing all of EquationElement’s abstract methods, it has its own constructor which takes in an integer. This value is stored in its value attribute and returned by its getValue method.

### Control Package

The control package contains classes for interfacing with the user and evaluating the user-provided equation.

#### Class Diagram

A diagram of a program

Description automatically generated

#### Parser Class

The Parser class parses the user-provided equation. It has one method: parse – a static method which takes in a string and parses from it a list of equation elements. In addition to parsing the equation, this class is responsible for detecting and reporting Invalid Symbol and Unmatched Parentheses errors (see the Software Requirements Document).

#### Evaluator Class

The Evaluator class calculates the value of the user-provided equation. Its lone method ‘evaluate’ takes in a list of equation elements and outputs the value of the equation that the list represents. In addition to evaluating the equation, the Evaluator class is responsible for reporting Missing Operator errors (see the Software Requirements Document).

#### Driver Class

The Driver class is the system’s main class. It interfaces with the user and drives the Parser and Evaluator. Its getUserEquation method prompts the user for an equation and returns their input. Its evaluateEquation method takes in a string – such as one generated by getUserEquation – and runs the Parser on it, and then runs the Evaluator on the Parser’s output, and then conveys to the user the Evaluator’s output – be that a value or an error. It run method drives these two methods, repeatedly calling each of them until the user indicates that they are done.

# 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]

# Size and Performance

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

# 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.]