Jayhawks

Arithmetic Expression Evaluator
Software Architecture Document

Version 1.0

Revision History

Date	Version	Description	Author
10/11/2023	1.0	The initial Software Architecture	Alexandra, Deborah, Riley, Timo, Victor, Ellia

Table of Contents[1]

[1]

Software Architecture Document

1. Introduction

This document contains our program's architecture. This will serve as a reference for our implementation. The definitions, acronyms, abbreviations, and references that will be used will be found in the introduction. The rest of the document will be constituted of the following subsections: Architectural Representation, Architectural Goals and Constraints, Use-Case View, Logical View, Interface Description, Size and Performance, and Quality.

1.1 Purpose

The purpose of the Software Architecture is to serve as a guide to implementation for the calculator application.

The following people use the Software Architecture:

- The **project manager** uses it to plan the project schedule and resource needs, and to track progress against the schedule.
- **Project team members** use it to understand what they need to do, when they need to do it, and what other activities they are dependent upon.

1.2 Scope

This *Software Architecture* applies to the Arithmetic Expression Evaluator. The Arithmetic Expression Evaluator is a simple expression calculator that processes user entered expressions. This document influences this project and this project only.

1.3 Definitions, Acronyms, and Abbreviations

Not Applicable

1.4 References

- EECS348: Term Project in C++ (00-Project-Description.pdf), Fall 2023, Published by EECS 348
    - Can be obtained on Canvas.
    - This reference will describe the project objectives and tasks, and different valid and invalid expressions.
- Arithmetic Expression Evaluator Software Development Plan (Software_Development_Plan_9_13_2023.pdf), 9.13.2023
    - Document can be found here:

      https://github.com/t878a585/348_Project/blob/main/Artifacts/Software_Development_Plan_9_13_2023.docx
    - This reference will describe the vision of our program.
- Arithmetic Expression Evaluator Software Development Plan (Software_Requirements_Specifications_10_14_2023.pdf), 10.14.2023
    - Document can be found here:

      https://github.com/t878a585/348_Project/blob/main/Artifacts/pdf/Software_Requirements_Specifications_10_14_2023.pdf

- This reference will describe the requirements of our program.

1.5 Overview

This *Software Architecture* contains the following information:

Architectural Representation — This describes the software architecture for our software and contains a diagram. This is just an overview of our architecture.

Architectural Goals and Constraints — This section will outline the software requirements and objectives that have some significant impact on the architecture.

Use-Case View — This discusses use cases from our use-cases model that illustrates important parts of our architecture.

Logical View — This view focuses on the internal logical structure of the application, emphasizing the organization of the components and their interactions.

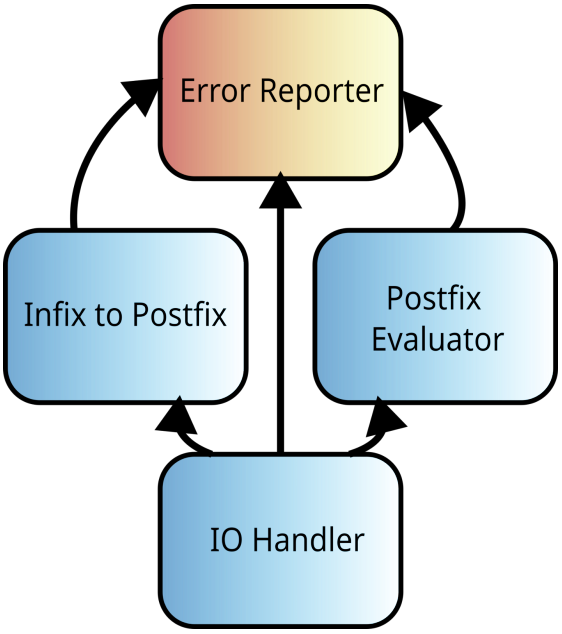
Interface Description — This describes the interface that the user will interact with.

Size and Performance — This refers to the storage and memory that the application consumes. Performance describes reducing load time and ensuring that the app runs smoothly.

Quality — The architecture quality describes functionality, usability, design, and reliability.

2. Architectural Representation

Our architecture is represented by the diagram below. Classes that are dependent on another class have an arrow pointing towards the class it is dependent on.



IOHandler instantiates the InfixToPostfix, PostfixEvaluator, and ErrorReporter classes. IOHandler then takes input and passes it to InfixToPostfix and then takes the output of that and passes it to PostfixEvaluator to get the resultant value of the expression. Both InfixToPostfix and PostfixEvaluator get an instantiated ErrorReporter passed to them. So, if there are any errors while processing data in this pipeline, they can be reported. If IOHandler sees that an error has occurred, it stops the pipeline and returns the error instead of the evaluated value.

3. Architectural Goals and Constraints

One of the main reasons that we are choosing to use a modular architecture is so that we can efficiently divide the work. If we were to make a large monolithic program, too much time would be wasted communicating and coordinating each other's work.

To make efficient use of the little time that we have, we also chose to forgo more complicated architectures (creating and evaluating a parse tree) and go with an architecture that allows us to process information with stacks.

There are other constraints that made us pick this architecture, but these two were the most important.

4. Use-Case View

Not Applicable

4.1 Use-Case Realizations

Not Applicable

5. Logical View

5.1 Overview

There will be five classes that make up the program. They are as follows:

ErrorReporter (ErrorReporter.hpp)

All errors from the InfixToPostfix and PostfixEvaluator modules are reported here. These errors are then collected by the IOHandler.

Token (Token.hpp)

Token is a class used to represent operands and operators abstractly.

InfixToPostfix (InfixToPostfix.hpp)

This takes a string in infix form and converts it to postfix form. This returns the postfix form as tokens.

PosfixEvaluator (PostfixEvaluator.hpp)

This takes tokens in postfix form and evaluates the value of the expression.

IOHandler (IOHandler.hpp)

This is the only class that handles IO. It also instantiates ErrorReporter, InfixToPostfix, and PostfixEvaluator. Once it receives an expression from the user, it passes it to InfixToPostfix, and then to PostfixEvaluator. If any errors are reported at any time, it reports the error and stops expression processing.

main.cpp

This instantiates IOHandler and calls its execution method.

5.2 Architecturally Significant Design Modules or Packages

Further details about each component are provided here.

5.2.1 *ErrorReporter (ErrorReporter.hpp)*

This class will consist of four public methods:

void add_error(char * module_Name, char * error _Description	When an object needs to report an error, this should be called. Module name is the name of the class reporting the error. Error description describes the error.
int get_error_count();	Returns the number of errors detected.
void remove_errors();	Clears the errors that are currently stored.
char * get_error_string(int index);	This returns the string representing the error. The returned string shall be freed by the callee.

The add_error() method will only be called InfixToPostfix and PostfixEvaluator. Every other method should only be called by IOHandler. IOHandler uses the three other methods to report errors.

5.2.2 *Token (Tonken.hpp)*

This class represents the individual components of an expression (operators and operands). Given the relative simplicity of this class, it has already been defined:

Token(long double _operand_Value);	When instantiating a token that represents an operand, this constructor will be used. The value of the operand is represented by a long double.
Token(char _operator_Value);	When instantiating a token that represents an operator, this constructor will be used. The value of the operator is represented a single character (‘/’,’+’,’-‘, etc.)
bool is_This_An_Operator();	If this is true, this token represents an operator, otherwise, it represents an operand.
char get_Operator();	If the token is representing an operator, this method is used to get the type of operator.
long double get_Operand();	If the token is representing an operand, this method gets the floating point value of the operand.

Whenever tokens are used, they will be passed using the std::list<Token> data type. In a list, they represent an expression from left to right.

Tokens are generated by the output of InfixToPostfix and consumed by PostfixEvaluator.

5.2.3 *InfixToPostfix (InfixToPostfix.hpp)*

This class will only have one public method:

std::list<Token> convert(char * infix_expression, ErrorReporter * error_reporter);	This takes in a string representing an infix expression (infix_expression). There should not be any newlines in the passed string. An ErrorReporter object should be passed (error_reporter). This should report an error if the infix expression is

	syntactically incorrect.

The purpose of this class is to convert infix to postfix notation and to check syntax, nothing else. This will use a stack to convert infix to postfix notation.

5.2.4 *PostfixEvaluator (PostfixEvaluator.hpp)*

This class will only have one public method:

long double evaluate(std::list<Token> postfix_expression, ErrorReporter * error_reporter);	This takes in a list of tokens (postfix_expression) in postfix form and an ErrorReporter object (error_reporter). It returns the numerical value of the expression passed in. If there are any mathematical errors (divide by zero, etc.), an error will be registered with error_reporter. The return value will be undefined in that case.

The purpose of this class is to evaluate the value of a postfix expression and report mathematical errors, nothing else. This will use a stack to evaluate the expression.

5.2.5 *IOHandler (IOHandler.hpp)*

This class will only have one public method:

void execute();	This executes the main execution loop for the entire program. Main calls this.

IOHandler handles most of the execution of the program. It gives an informational message at the start (how to use crt-c, Q or q to quit; about information; etc.), then it enters a prompting loop. It asks the user to enter an expression. Upon hitting enter, it calculates the value of the expression and returns it to the user. If after calling <name_of_InfixToPostfix_object>.convert() or <name_of_PostfixEvaluator_object>.evaluate(); the ErrorReporter object indicates an error, the evaluation of the expression will be immediately stopped and the error will be printed. Afterwards, it enters the prompting loop again.

To be able to do all of this, a few things need to be done. The first thing that IOHandler does is instantiates an ErrorReporter, InfixToPostfix, and PostfixEvaluator object. After having done this, it enters the prompt loop. After receiving user input, IOHandler checks to make sure there is no upper or lower case ‘q’ present anywhere in the string. If there is, the program ends. If there isn’t a check is then performed to make sure the string isn’t empty (a string with only newlines and/or white space). If the string is empty, the user prompt loop restarts. Otherwise, the expression is processed.

Before processing the string, IOHandler strips the newline at the end of the line. Once it has done that, it calls the convert() method of the InfixToPostfix object and collects the returned list of tokens. The methods of the ErrorReporter object are then used to determine if an error occurred. If one did occur, the processing of the expression stops, the error(s) are printed, and the loop starts again. Otherwise, these tokens are passed to PostfixEvaluator’s evaluate() method. If error occur here, the same action is taken as previously. If errors didn’t occur here, the value of the expression is printed and the loop starts all over again.

5.2.6 *main.cpp*

Main only instantiates IOHandler and calls its execute() method.

6. Interface Description

Once the program starts it will print the following message:

“

Welcome to the Arithmetic Expression Evaluator

To exit, type q or Q at any time and hit enter.

Alternatively, press control-c.

Whenever prompted with “Input your equation:”,

you may enter in your equation and hit enter to evaluate it.

“

Afterwards, it will prompt the user to enter an equation (“Input your equation:”). Once the user inputs their equation the code will output the answer. If there is a mathematical or syntactical error in their expression, an error will be printed indicating which object type reported it and what the error is. The program prompts after each error or evaluation for another expression until the program is terminated. The program can be terminated by typing Q or q and pressing enter or by pressing control-c by itself.

7. Size and Performance

Not Applicable

8. Quality

Various non-functional aspects of this software are determined by its software architecture. The architecture’s modular design makes it simple to add new functionality, easier to locate bugs, and easier to reuse components for other purposes. Given the small scope of our project these benefits may not matter much to us.