HEAAP Software

Arithmetic Expression Evaluator in C++
Software Architecture Document

Version <1.1>

Revision History

Date	Version	Description	Author
11/06/24	1.0	Initial Meeting and assignment of parts; discussion on architecture selection	All
11/10/24	1.1	Finalized Software Architecture Document	All

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

The purpose of the Software Architecture Document is to create a blueprint of the software system, including its design and development. Most importantly, this document guides development by providing a clear understanding of the system's structure, which, in turn, helps developers understand the work being done and avoids financial losses. Additionally, this document is a valuable resource for communication because it bridges gaps in understanding and serves as a reference for stakeholders. Key decisions are also highlighted and noted, which will aid in addressing future problems, as there will be a document to reference every critical decision. This document provides a structured approach for highlighting and organizing the main components, relationships, and decisions on architectural designs.

1.2 Scope

The Software Architecture Document influences many aspects of software development. Within the development process, code structure and quality, team alignment, scalability, and extensibility are greatly impacted. This document guides developers on code structure and ensures that quality is maintained. Since it contains the architecture of the system, it allows other teams, if needed, to work alongside, especially on larger-scale projects. Design and technical decisions are also impacted, as this document provides clear direction on how to design each component and how they should interact. The primary influence of this document is on stakeholders, as it clarifies how user needs are met and communicates limitations and design decisions. This document influences many aspects throughout the system's development that are essential in creating the software's structure.

1.3 Definitions, Acronyms, and Abbreviations

See Glossary in Appendix

1.4 References

See References in Appendix

1.5 Overview

The remainder of the Software Architecture Document is organized into sections that include Architectural Representation, Architectural Goals and Constraints, Logical View, Interface Description, and Quality. The first section defines Architectural Representation and the purpose it serves. The second section builds on the previous one by detailing the main goals and constraints of the system's architecture, including safety, security, and privacy. The Logical View communicates the significant architectural components, such as classes and their contents. The Interface Description section provides descriptions of the interfaces to be used, including screen formats, valid inputs, and resulting outputs. The Overview provides an overall description of the design that will be built. The final section, Quality, describes the system's extensibility, reliability, and portability.

2. Architectural Representation

The initial version of the system is being built using a main-program-subroutine software architecture. The components in this architecture are the subroutines which are helper functions used in the main function. The connectors in this procedural architecture are the subroutine function calls located in the main function of the program. The function calls are configured in an intentional manner that highlights the relationship between the components. The main function sits at the top of the hierarchy with access to the three subroutines. The evaluator subroutine depends on the parser subroutine which depends on the lexer subroutine. The subroutines do not depend on the main function and the lexer function can be thought of as the lowest in the hierarchy due to it being the first subroutine called. This architecture was selected due to the clearly defined and connected functional requirements and the small scale of this project. The user-interface view contains the lexer function which takes in the user input and processes it for parsing. This view also contains the main function's input/output functionality and user-friendly formatting. The functionality view contains the error handling measures implemented in all subroutines as well as the algorithm used to evaluate the correctly entered expression.

3. Architectural Goals and Constraints

Maintainability: Due to the use of subroutines as components, subroutines can easily be modified without having to make changes to the other subroutines.

Efficiency/Scalability/Performance: The architecture is designed in a manner that will be able to handle complex expressions and evaluate them without a significant decline in performance. Efficient algorithms will be housed within the subroutines.

Portability/Reuse: The software will be written in a clear format that can be seamlessly transferred to other environments if needed. Standard libraries will be used, and the main-program-subroutine architecture's simplicity is applicable across many languages in case the program is ever reused in a different language.

Constraints: Some subroutines such as the evaluator may contain many lines of code which can become confusing when distributing or passing off work to another team member. To combat this, clear and frequent commenting will be implemented throughout the subroutines as well as intentional variable naming. Mathematical rules and edge cases must be considered during the implementation to ensure the product is functional. The C++ libraries and codebase are also a constraint that will affect the implementation of this architecture.

4. Use-Case View

4.1 Use-Case Realizations

5. Logical View

5.1 Overview

The design model being implemented for this project is the main-program-subroutine architecture. A hierarchy of functions will be created where the main function has control over the subroutines (helper functions). These subroutines will be functions for lexing, parsing, and evaluating (error handling will be implemented within each of the subroutines but will not be a subroutine itself).

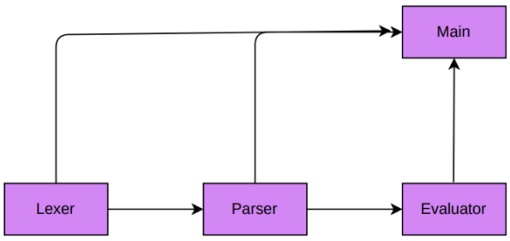
5.2 Architecturally Significant Design Modules or Packages

Lexer – Takes in expression entered by user as a parameter and first error checks to determine if it is a valid expression. Then, processes and stores the expression in preparation for the Parser (next subroutine).

Parser – Takes in an error checked expression as a parameter and parses the entire expression. The parser will also take care of handling parentheses in preparation for the evaluator. Returns an expression formatted in a manner readable by the evaluator function.

Evaluator – Takes in the formatted and parsed expression as a parameter and evaluates smaller expressions iteratively until a single double remains which is returned. Will have support for +, -, *, /, %, and ** operators and respect the rules of PEMDAS and left-to-right reading of the expression.

Main – Main function at the top of the hierarchy that calls the subroutines in a procedural order and handles obtaining user input as well as error checking that is not handled within the subroutines. Formats outputs to be compatible with the user-friendly interface in the command line.



6. Interface Description

The application will use a command line interface. The user will be prompted for an expression as a String. The following are valid operators: {+, -, *, /, %, ^, (,)}. The result will be output to the console as a double with three digits of precision after the radix point. The user will continue to be prompted for expressions to be evaluated until they input “exit” into the CLI.

7. Size and Performance

8. Quality

The program should be consistent and reliable in all is designed to do. The program will be distributed as C++ source files, meaning it is to be compiled on and for every target. This program does not handle sensitive data, so security and privacy are not concerns for this project.

9. Appendices

9.1 References

Bartelli, A., Barybin, P., Das, A., Farley, E., & Nurnberg, H. (2024). *Software Development Plan: Arithmetic Expression Evaluator in C++* (Version 1.2). LEEP, University of Kansas. Confidential © HEAAP Software 2024.

Bartelli, A., Barybin, P., Das, A., Farley, E., & Nurnberg, H. (2024). Software Requirements Specification: Arithmetic Expression Evaluator in C++ (Version 1.1). LEEP, University of Kansas. Confidential © HEAAP Software, 2024

Saiedian, Hossein. Software Architecture and Design Concepts. EECS 348: Software Engineering, Fall 2024. University of Kansas.

9.2 Glossary

Architectural Goals and Constraints: Requirements and objectives impacting architecture, such as safety, security, privacy, and reuse, and any specific constraints like design strategies and development tools.

Architectural Representation: A description of the architecture, including the views and elements that represent it.

Design Model: A representation of the architecturally significant parts of the system, decomposed into subsystems and packages, and detailing significant classes and their relationships.

Evaluator: A subroutine function that processes parsed expressions to return a single evaluated result, following the PEMDAS order of operations.

Interface Description: An outline of the major interfaces of the application, including input formats, valid inputs, and expected outputs.

Lexer: A subroutine that checks and processes user input expressions for validity before passing them to the parser.

Logical View: A section that outlines the significant components of the design model, including classes, their responsibilities, relationships, and hierarchy.

Main Function: The primary function that calls subroutines, handles user input, and manages general error checking outside of individual subroutines.

Overview: A description of the document’s organization and the content of each section, detailing the general structure and purpose.

Parser: A subroutine that processes the validated expression to handle parentheses and formats it for evaluation.

Quality Attributes: Non-functional requirements like extensibility, reliability, and portability that contribute to the system's overall capabilities.

Scope: The impact of the document on software development, including its influence on code structure, quality, and stakeholder communication.

Software Architecture Document: A blueprint of the software system, outlining its design, structure, and development guidance.

Use-Case Realizations: Scenarios demonstrating how specific design elements contribute to the system's functionality, often illustrated with select cases.