<Git Goblins>

Calc-U-Later
Software Architecture Document

Version 1.0.0

Revision History

Date	Version	Description	Author
12/11/23	1.0.0	Software architecture describing the overall design of the project. Added functionality.	Sabeen, Anna, James, Daniel, Kaden, Brett

Table of Contents

1. Introduction 4
   - 1.1 Purpose 4
   - 1.2 Scope 4
   - 1.3 Definitions, Acronyms, and Abbreviations 4
   - 1.4 References 4
   - 1.5 Overview 4

2. Architectural Representation 4

3. Architectural Goals and Constraints 4

4. Use-Case View 5
   - 4.1 Use-Case Realizations 5

5. Logical View 5
   - 5.1 Overview 5
   - 5.2 Architecturally Significant Design Packages 5

6. Interface Description 6

7. Size and Performance 7

8. Quality 7

Software Architecture Document

1. Introduction

This document will outline the comprehensive structure of the Calc-U-Later program, detailing the arrangement of software components, server organization, and the technologies utilized in the software such as C++, development process (as presented in the slides) and detailed in the version history. It will explore the specifics of software components, connectors, and configuration (topology). Also, it will offer a rationale for the decisions regarding the decomposition, identification, or definition of these elements. All of this is for the Calc-U-Later produced by Git Goblins which includes error handling, user input, and an algorithmic approach to a calculator.

1.1 Purpose

The purpose of this Software Architecture Document (SAD) is to provide a comprehensive architectural overview of the Calc-U-Later Application. It captures the significant architectural decisions, design principles, and the structure of the application. This document serves as a reference for development teams and stakeholders, facilitating communication and understanding of the system's architecture.

1.2 Scope

Scope is defined as the domain over which a pattern applies. Regarding our calculator, the scope would entail a calculator that a user has the ability to input basic arithmetic including parentheses and including error handling. This document influences the way that the code was written to ensure that it is developed in a dependable, secure, and efficient manner. This ensures the software comes with well-structured and organized components and includes the necessary technologies to ensure that it is completed correctly.

1.3 Definitions, Acronyms, and Abbreviations

- Software Architecture: to create a reliable, secure, and efficient product, you need to pay attention to architectural design (1).
- Architectural Design: includes factors such as overall organization, how the software is (1). decomposed into components, the server organization, and the technologies used to build the software (1).
- Component: encapsulates a subset of the system’s functionality and/or data (1).
- Decomposition: break up components into smaller, manageable parts (1).

1.4 References

1. *Software Architecture and Design Concepts* Lecture by Professor Hossein Saiedian; Slide 6-9
2. Calc-U-Later Git Goblins Project Development Plan Document v1.0
3. Calc-U-Later Git Goblins Software Requirements Specifications v1.0
4. *Layered Architecture* by University of Waterloo

1.5 Overview

The following sections include the architectural representation, the goals and constraints, a use-case view, , logical view, an interface description, quality, and size and performance as it relates to the Calc-U-Later. The document is organized in this way so we can first demonstrate how it is used, such as the expression parser, and evaluator, and how this affects our scope. Moreover, the rest of the document follows as such to explain the rational and quality of our decisions and how it affects the performance of the Calc-U-Later

2. Architectural Representation

Functional view: the functional view encompasses various model elements to represent the functional elements of the software. This view is made up of functions (which represent specific operations the system can perform), data structures (which represent the organization of data within the system), components, interfaces, variables and use-cases.

Structural view: the structural view emphasizes the organization and relationships between the software's components, functions and data structures. This view helps you understand how the software is structured. For example, it could reveal how the user interface interacts with the mathematical calculation functions.

Logical views: The Logical View of the current system's software architecture provides an internal perspective of the system, emphasizing its structural organization and the interactions among its components. This view focuses on how the system's components, classes, and packages are structured and how they collaborate to achieve the system's functionality. It helps developers and architects understand the system's internal composition and relationships. In our program, we will be using different packages like User Interface, Expression, and Error Handling. These packages consist of different classes. More information can be found in section 4.2.

3. Architectural Goals and Constraints

One of the major goals of our program is to make sure that our program is secure. We intend on doing this through adding multiple try/catch blocks so that we can hopefully catch any errors that occur in our code. While it would be easier to simply have one try/catch block, we believe that it's important to give a truly detailed and impactful error to the user, and the way to do this is to have multiple spots where error handling will be practiced throughout the program, and, in addition to that, this also implies that our program will be segmented into different parts. Ensuring that different components of our program are kept separate adds to the maintainability of our program. The potential for portability in our calculator is immense since anyone could write a function for any mathematical equation, and they'd be able to simply call the functions that we have already built to add it into our code.

4. Use-Case View

4.1 Use-Case Realizations

5. Logical View

The Logical View of the "Calc-U-Later" calculator application focuses on the architecturally significant parts of the design model. It describes the decomposition of the system into subsystems and packages, introducing architecturally significant classes, their responsibilities, and important relationships, operations, and attributes.

5.1 Overview

This Logical View outlines the key subsystems, packages, and architecturally significant classes of the "Calc-U-Later" calculator application. It focuses on fulfilling the specific requirements of expression parsing, evaluation, operator precedence, parentheses handling, and error handling while maintaining a user-friendly interface for entering expressions. These components collectively define the logic and behavior of the application.

5.2 Architecturally Significant Design Modules or Packages

1. User Interface Package:

Description: Responsible for providing user interface so they can input an expression

Architecturally Significant Classes:

- UserInput: Allows user to enter an expression and stores it into the system to be evaluated.

2. Expression Package:

Description: Responsible for parsing arithmetic expressions and evaluating.

Architecturally Significant Classes:

- ExpressionParser: Parses the input expression and generates an expression tree.
- ExpressionEvaluator: Responsible for applying the different operator methods to evaluate the given expression.

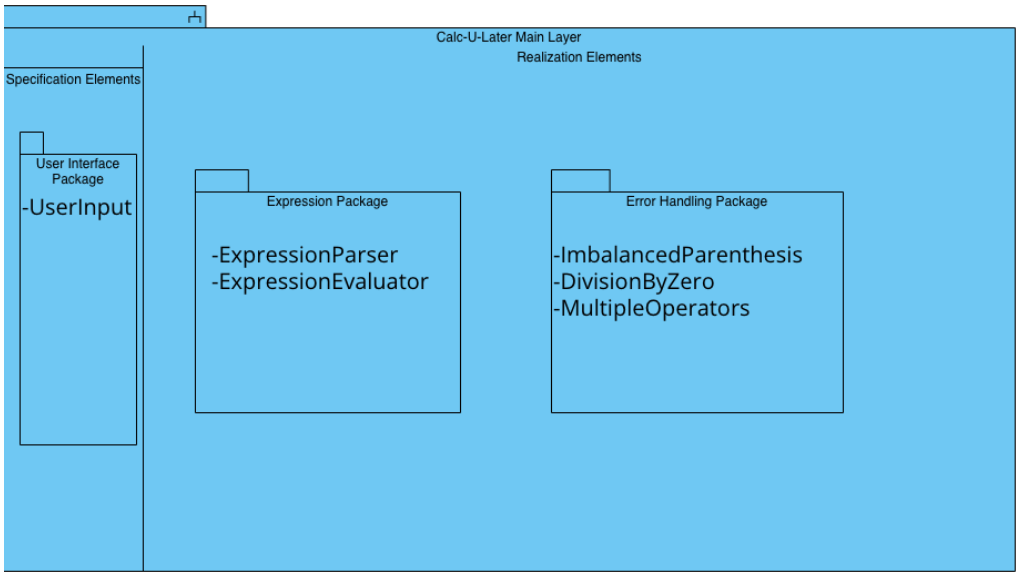
3. Error Handling Package:

Description: Manages any errors or invalid expressions that need to be handled.

Architecturally Significant Classes:

- ImbalancedParanthesis: Handles imbalanced parenthesis
- DivisionByZero: Handles any expression where zero is in the denominator
- MultipleOperators: Handles when multiple operators, that can not be placed back to back, are next to each other, example: */

Calc-U-Later single layer architectural design:



6. Interface Description

Our major entity interface is in the form of a command line interface. It is always going to appear with the same background, and it will remain consistent. Valid inputs consist of possible mathematical operations such as addition, subtraction, modulo, multiplication, exponentiation, and division. The end-user of the Calc-U-Later will interact with a command line interface inputting numbers, operators, and parenthesis as such and if an error were to occur, they would be prompted that they input an error and would have to

reenter their expression.

7. Size and Performance

8. Quality

The software architecture contributes to all capabilities other than functionality by allowing the program to be better updated and maintained in the future. Without our compartmentalization of key functionalities, we wouldn't be able to easily dissect where we're having issues as we currently are able to. If our calculator is outputting integers instead of floats for one function, then we can know which function to go to. If our calculator is struggling to parse out parenthesis, then we can go to that part of the calculator and examine why our parenthesis parser isn't working as previously intended. Moreover, if our program crashes when doing division by zero, there's a clear function that we can go to in order to address this problem. This holds special significance in preserving the integrity of our program through error handling. In addition, because our functions utilize other functions and aren't built as a monolith, the extensibility of the functionality of our program is immense. If we chose to do complex algebra or something in the future, we can use the functionality we've already built out to do so instead of starting all over again.