Team 2

Calculatorator
Software Architecture Document

Version 1.1

Revision History

Date	Version	Description	Author
11/7/2024	1.0	Added intro	Riley Anderson
11/10/2024	1.1	Added goals/constraints	Kaia Burkholder
11/11/2024	1.2	Added section 5.1 and 5.2	Sam Kelemen

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

The purpose of this Software Architecture document is to outline the object-oriented programming style in which we chose to implement the Calculatorator. Within this document, we detail exactly which ‘objects’ exist and how they interact with the greater environment of the software

1.2 Scope

This Software Architecture Design document describes the architecture that will be used to build and implement the project. The details of the individual iterations will be described in the Iteration Plans.

The plans as outlined in this document are based upon the product requirements as defined in the Project Plan, and include the program properly handling +, -, *, /, %, and **, processing numeric constants, managing parentheses, and handling errors like division by zero.

1.3 Definitions, Acronyms, and Abbreviations

OOP: Object oriented programming, a computer programming model that organizes software design around data, or objects, rather than functions and logic.
QA: Quality Assurance

1.4 References

-Iteration Plans: Not yet created

-Vision: Create a fully functioning, robust calculator developed entirely within C++. The program must properly handle +, -, *, /, %, and **, process numeric constants, manage parentheses, and handle errors like division by zero.

1.5 Overview

This Software Architecture Document is responsible for laying out all necessary information to effectively implement the OOP nature of the software. It begins by detailing the representation of the architecture, laying out our goals, and then tackles the problem in a more practical way. From there we describe the interface that we will provide, and then how we will assure quality of the software product.

2. Architectural Representation

The software architecture is based on OOP for the calculator. The system has been represented in a Use-Case view (which can be found in the requirements document) and a logical view. The logical view will break down the program into separate files and the classes and functions within those files.

3. Architectural Goals and Constraints

The initial implementation of the calculator is set to be due on December 8th. The program needs to follow the principles of object-oriented programming, limiting how the program can be structured. The program also requires error handling, determining how the different parts of the program need to connect and break when errors occur.

For general requirements, the program has to follow PEMDAS in its calculations of given equations. This can determine how the data is structured. Its is also possible that the project will have to implement compatibility with floats, which can change the ways the inputs are saved and how the answer is given back

to the user.

4. Use-Case View

4.1 Use-Case Realizations

5. Logical View

5.1 Overview

The overall program will be structured as follows:

cmd_parser_with_tests/

```
├── cmd_parser/
│   ├── parser.cpp
│   ├── calculator.cpp
│   ├── main.cpp
│   └── user_interface.cpp
└── tests/
    ├── test.cpp
    └── test_cases.txt
```

The *command_parse_with_tests* directory contains the *cmd_parser* directory, which houses the *main* program, and the *tests* directory, which contains a script to test the program as well as a text file that contains the individual test cases to run. The *main* module is the entry point to the program. It will call functionality from the *user_interface* module to interact with the user. The *main* function will then pass strings from the user to the *parse* module, which will pass a list of commands to the *calculator* module, which will execute these commands and return the result back to the *parser* module and then back to the *main* function to handle presenting the result to the user via the command line.

5.2 Architecturally Significant Design Modules or Packages

‘cmd_ln_parser’ Package:

cmd_parser/

```
├── parser.cpp
├── calculator.cpp
├── main.cpp
└── user_interface.cpp
```

parser.cpp : The *parser* module will parse the input string into a list of commands to be executed by the *calculator* module if the input string is a valid expression. Otherwise, it will return an error code to the main module.

calculator.pp : The *calculator* module will execute the commands fed to it from the *parser* module.

main.cpp : The *main* module will be the program that is executed to run the program. The *main* module will call the *user_interface*, *parser*, and *calculator* modules to perform the functionality of the program.

user_interface.cpp : The *user_interface* module will handle all input and output from the command line. It will interact with the *main* module to read in input and print the appropriate messages to the command line to facilitate the user experience.

‘tests’ Package:

tests/
├── test.cpp
└── test_cases.txt

test.cpp : The *test* module is not part of the final product. It will contain a class to execute a list of test cases stored in the *test_cases.txt* file. The *test* module should interact with the *main* module to execute each of the test cases. Additionally, it should print out to the command line the state of each of the tests (whether they passed, and if not, the incorrect result of the program and the correct result to compare).

test_cases.txt : The *test_cases.txt* module will contain a list of test cases, and their solutions, to test the complete functionality of the program. Each test case will be on a new line so that the *test* module can easily execute each test case line-by-line

6. Interface Description

User Interface:

The calculator will operate like a command line for the user. The user will input expressions to the terminal, and the output will be displayed in the terminal also. Here are the following formats

1. Input Format
    a. The system will provide a prompt for the user to enter an expression, for example:
        i. Enter an expression:
    b. Valid Input: Use can input mathematical expressions in standard form (+ as addition, - as subtraction, * as multiplication, / as division, % as modulo operator, ** as exponent operator) Parenthesis also supported
    c. An example input:
        i. 2 + 2
        ii. (5 + 1) / 3
        iii. 8 / 2 + 4
    d. Invalid inputs such as dividing by zero, unbalanced parentheses, or unsupported characters will be flagged as errors
2. Output Format
    a. The program will display the result in the format:
        i. Answer = 8 (if the expression is 5 + 3)
    b. In case of errors, such as division by zero or syntax issues, the program will print an error message like this:
        i. Error: Division by zero
        ii. Error: Invalid syntax
3. Module Interactions

The interactions between the modules like main.cpp, parser.cpp, calculator.cpp, user_interface.cpp can be summarized as:

a. main.cpp to user_interface.cpp:
    i. The main.cpp file calls the user_interface.cpp to collect user input and to display results.
    ii. The main.cpp passes user input as a string to the parser.cpp module for parsing.
b. user_interface.cpp to parser.cpp:
    i. The user_interface.cpp provides the raw input from the user (a string) to parser.cpp.
    ii. The parser.cpp returns a list of commands or a result that the user_interface.cpp will display.

c. parser.cpp to calculator.cpp:
    i. The parser.cpp passes a list of parsed commands to calculator.cpp, which then evaluates the mathematical expression.
    ii. The calculator.cpp returns the result or an error code to parser.cpp, which then returns it to main.cpp for display.
d. calculator.cpp Internal Interface:
    i. The calculator.cpp contains functions that implement the actual mathematical operations, such as add(), subtract(), multiply(), etc.
    ii. It uses helper functions for parsing numbers and operators, handling parentheses, and managing errors (like division by zero).

4. Sample Interfaces
    a. Enter an expression: 3+4
    Answer = 7
    b. Enter an expression: (3+1)*6
    Answer = 24
    c. Enter an expression: (5-8) / 3
    Answer = -1

7. Size and Performance

8. Quality

The software architecture is designed to support important non-functional capabilities like extensibility, reliability, and maintainability. By using OOP it makes it easier to add new features. The program has different parts, the calculator, parser and user interface to organize things and make it easier to troubleshoot errors. It also checks common mistakes like dividing by 0 or unbalanced parentheses. This makes sure that the program works well even when things do not go to plan. Since the program is made in C++ it can run on many different operating systems. The overall structure of the system makes it easy to update or fix bugs. While security is not a big concern, making sure there is correct input to the system ensures a safe system.