17 Co.

17 Co. Calculator
Software Architecture Document

Version 1.0

Revision History

Date	Version	Description	Author
11/Nov/23	1.0	Initial Software Architecture Spec	17 Co. Team

Table of Contents

Software Architecture Document

1. Introduction

The Software Architecture Document (SAD) outlines the Design phase of creating the *17 Co. Calculator.* To understand the document, you can see the purpose, scope, definitions, acronyms, abbreviations, references, and overview in the Introduction section. This section can be used as a guide to understanding the document as a whole.

This document was developed as a part of the Design phase. If a change in project architecture is to occur in a future phase, this document should be updated. You can tell the current state of the document based on the version number in the header of each page. If a complete change in the document is to occur the major version number should be incremented (i.e. v1.0 -> v2.0). If a small change in response to a new phase is to occur the secondary version number should increment (i.e. v1.0 -> 1.1). If a small change during a phase must happen a third version number should be implemented (i.e. v1.0 -> v1.0.1). Using this guide, you can tell the iterations of the document and potential changes that have occurred. When those changes occur, the changes will also be marked in the section entitled “Revision History” on Page 2.

1.1 Purpose

The purpose of the SAD is to document and solidify a design plan for creating the *17 Co. Calculator*. This includes the architecture of the program from a use-case perspective, a user interface (UI), a user experience (UX), and a logical perspective. This document should serve as a guide during development and implementation that should be able to answer most questions that arise during the process. The SAD can also be used by an end-user to get a better understanding of the architecture of using the program.

In addition, the Technical Lead should use this document to inform decisions in leadership during the upcoming implementation phase, and the Quality Lead should use this document once implementation has concluded to ensure that the final product matches the intended use-case, UX, and UI experience.

1.2 Scope

The SAD applies to the entirety of the design phase during the creation of the *17 Co. Calculator* and should be the main location for documentation surrounding it. Due to the close nature of the design phase and the implementation phase, what is described in this document will also apply to the upcoming implementation phase.

1.3 Definitions, Acronyms, and Abbreviations

Various terms, acronyms, and abbreviations may be used throughout this document, including some from previous development documents, which can be found in the *Project Glossary*. Some new terms, acronyms, and abbreviations can also be found below:

SAD: The Software Architecture Document. This document, describing the architecture of the project.

UI: User interface. The way the program presents itself to the user for both input and output.

UX: User experience. The way in which the user uses the program and experiences the features of 17 Co. Calculator.

IO: In-out. The concept of how input and output are handled.

1.4 References

Various sections of the SAD may reference other documents. You can see a list of all of them here:

PROJECT GLOSSARY – File ID D00002
SOFTWARE DEVELOPMENT PLAN – File ID D00001
SOFTWARE REQUIREMENTS SPECIFICATIONS – File ID D00003

1.5 Overview

This document is broken up into several sections to divide the architecture description into smaller more concentrated parts.

- Section 1: Outlines the high-level details of the SAD and gives context to the document within the project as a whole.
- Section 2: Describes the high-level view of the architecture of the program, breaking down the various ways in which the architecture can be designed.
- Section 3: Describes the goals and constraints within the architecture of *17 Co. Calculator* including how the architecture should be built to meet those goals and stay within those constraints.
- Section 4: Unneeded for *17 Co. Calculator.*
- Section 5: Describes the logical view of the project, looking at the architecture from the back-end perspective.
- Section 6: Describes the user's view of the project, looking at the architecture from an IO perspective.
- Section 7: Unneeded for *17 Co. Calculator.*
- Section 8: Describes the quality assurance aspects of the architecture and specific tasks that should be done to ensure a consistent experience.

2. Architectural Representation

Software architecture in this project is the structure by which the project will be developed and how the end-user will interact with the program. It is represented in the class structure, the function structure, and other such enumerations. The architecture of this system is fairly simple to understand, and it can be viewed through two main views, Use-Case and Logical. Both will be elaborated on further in this document. In summary, the use-case view will contain all elements that demonstrate how the program is structured and functions from the user's perspective. The logical view will contain the larger architecture and data flows from the perspective of basic system design.

3. Architectural Goals and Constraints

There are four main constraints on our architectural solution: ease-of-use, performance, security, and portability. The product must be simple and easy for a user to understand, which means the architecture must be designed in order to maximize, or at least facilitate the delivery of, a user experience that matches reasonable expectations from the user. These expectations may be set by other similar programs, for example. The program must also be designed with speed in mind, ensuring that using the program will be faster than simply hand-calculating results, and so the architecture must reflect that. Additionally, it must be secure. The architecture must be designed according to best practices and with security first and foremost. The program must also be portable, being able to be used in a wide variety of systems. The architecture should be as self-contained as is reasonable and allow the program to run in different environments.

Other architectural goals and constraints include the ability for the program to be developed in a reasonable timeframe by a small team. For example, the program architecture must allow for collaborative development to be done through the use of version control systems such as Git and repository hosting services such as GitHub.

4. Use-Case View

Unneeded by 17. Co Calculator.

5. Logical View

The software will all be written in one C++ file. The file will include the C++ standard input-output library “iostream” and the standard math library “cmath”. While it may not contain any custom classes, the program will have its own functions involving both recursive and iterative elements.

We will have a function that will take the user’s input and store it in a way that will be easy to manipulate during implementation. The current plan is to create a dynamic vector that contains the values and operations to be calculated.

There will be another function that will process the mathematical functions given by the user. This function will use a recursive solving algorithm to break the problem down and solve it step by step until the answer is returned.

A simplification system will also need to be implemented to make the recursive algorithm work as effectively as possible. It is extremely important that the recursive algorithm is implemented correctly as it will handle the entirety of the calculations.

5.1 Overview

The standard input-output library is necessary for the program to take input from the user and output a value, in this case, a calculated solution. It will allow the program to take a mathematical expression from the user, interpret the equation, calculate a solution, and then output the result to the user via the text-based interface. The standard math library is used since it provides additional mathematical functionality that is necessary for evaluating the mathematical expressions given by the user. Examples of these additional mathematical functionalities include exponentials and rounding properties.

5.2 Architecturally Significant Design Modules or Packages

The standard C++ input-output library is a library that allows C++ programs to receive various forms of input from the user.

The standard C++ math library is a library that contains functions necessary to perform various math operations. Some of these will be necessary for our program to evaluate the mathematical equations the user enters. For example, our program needs to be able to interpret and evaluate exponents. There would be no intuitive way to do this without the library, but the library contains the “pow()” function which performs the calculation given the base and the exponent.

6. Interface Description

The software’s interface will consist of a text-based system in which the user will enter their math query and the answer to the equation will be printed to the console.

The formatting will show the equation that the user entered as well as an answer below it. Then the console will skip a line before accepting the next query. Valid inputs will be any inputs in which there are any number of combinations of an integer with all acceptable extensions such as exponential powers or negatively signed followed by an acceptable integrated operator such as addition, subtraction, multiplication, etc. The input will also have to end with another integer so as not to have the query end with an operator.

The resulting output will be an integer that is returned from solving the equation in the program. For the first installment of this software, an acceptable output will be a correctly executed equation when given two integers.

7. Size and Performance

Unneeded by 17. Co Calculator.

8. Quality

The software architecture of the *17 Co.* Calculator contains both recursive and iterative elements. This should allow for great extensibility as future functions and features could be added as an iterative step without the need for a redesign. Unless there is a case in which there is a significantly demanding enough feature, this program should be able to be altered and improved with relative ease.

Additionally, the recursive elements of the software should provide a good indicator of the reliability of the function. With each step of the solving process relying on the same function, if there were an issue with the code, it should become apparent relatively early in testing. The only necessary step to ensure reliability is testing each iterative section of the recursive function. If all methods of recursing are working properly, then the function as a whole should work properly.

Another benefit of this architecture is its portability. With recursion being widely supported by most high-level programming languages, this function should be easy enough to move between many different software elements. Additionally, this architecture allows for the demands of the system to scale with the difficulty of the problem. This allows it to work, possibly with constraints, in even very limited environments.

One drawback, however, of this architecture is the need for additional error handling. Because there is the possibility of a larger than normal stack, we will need to prepare for stack overflow errors and other methods of managing recursive depth.

Given the nature of the software, we do not expect significant privacy, security, or safety implications.