Arithmetic Expression Webpage

Software Architecture Document

Version <1.0>

[Note: The following template is provided for use with the Unified Process for EDUcation. Text enclosed in square brackets and displayed in blue italics (style=InfoBlue) is included to provide guidance to the author and should be deleted before publishing the document. A paragraph entered following this style will automatically be set to normal (style=Body Text).]

[To customize automatic fields in Microsoft Word (which display a gray background when selected), select File>Properties and replace the Title, Subject and Company fields with the appropriate information for this document. After closing the dialog, automatic fields may be updated throughout the document by selecting Edit>Select All (or Ctrl-A) and pressing F9, or simply click on the field and press F9. This must be done separately for Headers and Footers. Alt-F9 will toggle between displaying the field names and the field contents. See Word help for more information on working with fields.] Marked (shaded) areas: items

that are OK to leave out.

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| <11/12/23> | <1.0> | Description of SAD | Zutshi team |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

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. Logical View 5

4.1 Overview 5

4.2 Architecturally Significant Design Packages 5

5. Interface Description 5

6. Quality 5

Software Architecture Document

# Introduction

The following will describe the purpose, scope, definitions, acronyms, abbreviations, references, and the overview of the Arithmetic Expressions HTML page being developed.

## Purpose

The aim off this SDP is to provide a comprehensive overview of the methodologies, development practices, and management strategies being used for the AE HTML page. All parties involved shall use this as a blueprint to engage in the project in the proper manner so that a structured, and unambiguous rollout of the product can occur. This can also be reflected on for when the document might need any kind of maintenance:

* 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.

## Scope

This SDP encompasses the creation of the AE (Arithmetic Expressions) HTML page. This will be a webpage capable of handling all PEMDAS expressions inputted by the user and returning the proper output after following the proper order of operations. It will also be capable of handling roots, otherwise known as fractional exponents.

## Definitions, Acronyms, and Abbreviations

* SDP: Software Development Plan
* AE: Arithmetic Expression
* PEMDAS: referencing common acronym used in arithmetic expressions to solve order of operation problems
* HTML: stands for “Hyper Text Markup Language,” it is the standard markup language used to create webpages
* CSS: stands for “Cascading Style Sheets,” stylesheet language used to format HTML doc, add color, explains how the render should look on the screen for the user interface
* PL: project lead
* FE: Front end
  + Front end development has to do with what you see and interact with on a website
* BE: Backend
  + Back end development has to do with the code that is running behind the scenes to make the front end functionable
* FEL: Front end lead
* BEL: Back end lead
* Push: committing a block of code to the main code repository
* Digital Ocean: an online cloud server host we will use to put our website available for all to see from their own hosts
* Oat++: open source C++ web framework available on github
* C++: coding language to be used on backend development
* JavaScript: code that can be integrated into an html page for the frontend
* Stakeholder: someone involved in the development process, whether it be directly working on development, funding development, or overseeing development
* POSIX(Portable Operating System Interface): family of standards specified by the IEEE for maintaining compatibility between operating systems
* cpp-httplib: a C++ library designed to handle HTTP client/server communication
* outbound connection: web application initiates a connection to another server or service on the internet
* inbound connection: When external entities, like users or other servers, initiate a connection to the web application
* OS Socket: Sockets are virtual "doorways" in a computer system, managed by the operating system, that let data in or out over a network. They operate using specific rules, known as an API
* API: Application Programming Interface, set of rules that regulates how different software’s exchange data with one another, can include for easier functionality

## References

* This subsection provides a detailed list of documents and resources cited within the SDP. Each document will be organized and recognizable by its title, report number, publication date, and/or the person who is responsible for its development. This way, if there is ever a problem regarding code integration or simply citations, this can be looked back at to resolve it.
  + Slides from EECS 388
    - Majority of plan used this as a reference
  + Iteration Plans
    - This is can be found in section 4.2
  + Digital Ocean
    - <https://www.digitalocean.com/go/developer-brand?utm_campaign=amer_brand_kw_en_cpc&utm_adgroup=digitalocean_exact_exact&_keyword=digitalocean&_device=c&_adposition=&utm_content=conversion&utm_medium=cpc&utm_source=google&gad=1&gclid=Cj0KCQjwvL-oBhCxARIsAHkOiu2-EyZwKVlsf-riHa1mgS0LzwrCJM2k6hu2t7dyAZwqQ_eQhOAK80MaAimsEALw_wcB>
  + Github
    - All our code will be stored on github, allows for everyone to pull and push code, (leads will approve code pushs)
    - <https://github.com/zmanna/SoftwareEngineering>

## Overview

The following document describes the Software Architecture that goes into the development of the Arithmetic Expressions Web Application.

# Architectural Representation

The architecture for the system can be broken down in four different ways: Logical view, Development View, Physical View, and Process View.

Logical View:

* Modules:
  + Expression Parser: The string input provided by the user will be able to be broken down into a structured format that the system can understand.
  + Calculation Engine: That structured format from the previous step will be taken and put through this module, this module has the capacity to carry out arithmetic functionality and will compute a result.
  + Result Formatter: The output still needs to be formatted for user readability, it will be cast back into a string and displayed on the user interface.
* Interfaces
  + User Interface: The user interface is the web page; the webpage has a GUI of a calculator and text boxes for the user to type into and see their results.
* **Development View:**
  + **Code Organization:**
    - **Frontend**: JavaScript files handling user input, validation, and display logic.
    - **Backend**: Python/Flask server with REST API endpoints for receiving expressions and returning calculation results.
    - **Common Libraries**: jQuery for DOM manipulation, Bootstrap for UI styling.
  + **Development Environment:**
    - **Version Control**: Git with GitHub for source code management.
    - **Build Tools**: Webpack for bundling JavaScript files, pip for managing Python dependencies.
* **Physical View:**
* **Deployment Configuration:**
  + **Web Server**: An AWS EC2 instance running Nginx to serve the static files and act as a reverse proxy for the Flask application.
  + **Application Server**: A Gunicorn server running the Flask application.
  + **Client**: Users' browsers (Chrome, Firefox, etc.) that render the webpage and execute the JavaScript code.
* **Networking:**
  + **HTTP(S) Requests**: Communication between client browsers and the web server over the internet using the HTTP/HTTPS protocols.
* **Process View:**
* **Runtime Behavior:**
  + **User Input**: The user enters an arithmetic expression into the webpage and hits 'calculate'.
  + **Request Handling**: JavaScript on the client side sends the input to the Flask backend via an AJAX call.
  + **Expression Evaluation**: The backend parses the expression, calculates the result, and sends it back to the client.
  + **Displaying Results**: JavaScript receives the result and updates the DOM to show the calculation output to the user.
* **Data Flow:**
  + **Input/Output**: Data flows from the client to the server and back as JSON objects. For example, **{“expression": "2+2”}** and **{"result": “4”}**.

# Architectural Goals and Constraints

The architectural goals have been described above, now the constraints must be discussed. There are many forms of constraints, some more applicable to this project than others, but we will go through all of them.

* **Safety**: The application implements client-side validation to ensure that only valid arithmetic expressions are processed, preventing the evaluation of potentially harmful code.
* **Security**: User input is sanitized to prevent injection attacks. Secure communication protocols (HTTPS) are enforced for data transmission to protect against eavesdropping and man-in-the-middle attacks.
* **Privacy**: No user data is stored persistently, adhering to privacy-by-design principles. Any temporary storage of expressions for the duration of the session is handled in-memory and is not logged or archived.
* **Use of Off-The-Shelf Products**: The system leverages well-established open-source libraries for parsing expressions and performing calculations to reduce development time and increase reliability.
* **Portability**: The application is built using cross-platform technologies (HTML, CSS, JavaScript) to ensure it can run on any operating system and device without modification.
* **Distribution and Reuse**: The architecture is modular, allowing components like the Calculation Engine to be reused in different contexts or distributed as standalone services if needed

# Logical View

In this section, the logical view will be focused on and broken down into its modules and interfaces, with in depth explanations for each part along with their respective class diagrams.

## Overview

Logical View:

* Modules:
  + Expression Parser: The string input provided by the user will be able to be broken down into a structured format that the system can understand.
  + Calculation Engine: That structured format from the previous step will be taken and put through this module, this module has the capacity to carry out arithmetic functionality and will compute a result.
  + Result Formatter: The output still needs to be formatted for user readability, it will be cast back into a string and displayed on the user interface.
* Interfaces
  + User Interface: The user interface is the web page; the webpage has a GUI of a calculator and text boxes for the user to type into and see their results.

## Architecturally Significant Design Modules or Packages

Logical View:

* Modules:
  + Expression Parser: The string input provided by the user will be able to be broken down into a structured format that the system can understand.
    - Class Diagram:
    - Classes:
      * Tokenizer:
        + Description: breaks down the input string into tokens
        + Responsibilities: token generation, categorization, and syntax validation
      * Parser:
        + Description: Converts tokens into a parse tree
        + Responsibilities: syntax tree generation, maintaining operator precedence, handling parentheses(this will be broken down into another separate function)
  + Calculation Engine: That structured format from the previous step will be taken and put through this module, this module has the capacity to carry out arithmetic functionality and will compute a result.
    - Class Diagram:
      * Classes:
        + Arithmetic Processor:

Description: handles arithmetic expression logic

Responsibilities: execute operations, manage order of operations, return intermediate results

* + - * + Operations: ‘add()’, ‘subtract()’, ‘multiply()’, ‘divide()’
        + Attributes: ‘operationQueue’, ‘CurrentResult’
  + Result Formatter: The output still needs to be formatted for user readability, it will be cast back into a string and displayed on the user interface.
    - Class Diagram:
    - Classes:
      * Numeric Formatter:
        + Description: Formats numerical results into a string with proper formatting
        + Responsibilities: decimal precision usage, needs to be able to apply locale formatting
        + Operations: ‘setPrecision(result, precision)’, ‘formatNumber(result)’
        + Attributes: decimalPrecision, locale
* Interfaces
  + User Interface: The user interface is the web page; the webpage has a GUI of a calculator and text boxes for the user to type into and see their results.
  + Calculator GUI
    - Design: The graphical user interface resembles a standard calculator with a grid layout of buttons for digits (0-9), operations (addition, subtraction, multiplication, division), and control actions (clear, equals)
    - Functionality: Clicking the calculator buttons triggers a JavaScript function that appends the corresponding character to a string displayed in a text box. This allows users to build and edit their arithmetic expression visually.
    - Interaction: The GUI captures mouse clicks on buttons and reflects the input in a text box. It also supports keyboard input for users who prefer to type expressions directly.
    - Class Diagram:
      * Classes:
        + Arithmetic Processor:

Description: handles arithmetic expression logic

Responsibilities: execute operations, manage order of operations, return intermediate results

* + - * + Operations: ‘add()’, ‘subtract()’, ‘multiply()’, ‘divide()’
        + Attributes: ‘operationQueue’, ‘CurrentResult’
  + Expression Display:
    - Textbox: A prominently placed textbox within the GUI shows the current arithmetic expression as it's being composed. The textbox content updates in real-time with each button click or keystroke.
    - Error Handling: If an invalid sequence of characters is entered (e.g., two consecutive operators), the system provides instant feedback, such as highlighting the error or displaying a warning message.
    - Class Diagram:
      * Classes:
        + Display Controller:

Description: controls error messaging and expression display in UI

Responsibilities: update the display expression, manage error feedback

* + - * + Operations: updateExpressionDisplay(expression), displayError(message)
        + Attributes: currentExpression, errorMessage
  + Hyperlinks to Additional Resources:
    - Description: Offers quick access to educational content, helping users understand the principles behind the operations and the project itself
    - UML Diagram of our project: A UML diagram page showing all the relations between classes and actors done in the UML format we have been practicing.

# Interface Description

* **Calculator GUI**:
  + **Screen Format**: The screen displays a calculator interface with a numeric keypad including buttons for numbers 0-9, operations (addition, subtraction, multiplication, division), and control actions (clear, equals).
  + **Valid Inputs**: The GUI accepts mouse clicks on buttons, touch input on touch-screen devices, and keyboard inputs for numbers and operations. Valid inputs include any sequence of numbers and arithmetic operators that can form a legitimate arithmetic expression.
  + **Resulting Outputs**: The output is the arithmetic expression displayed in real-time in the textbox as it is being entered. When the 'equals' button is pressed or an equivalent action is taken, the calculated result is displayed in the same textbox.
* **Expression Display**:
  + **Screen Format**: A textbox is presented at the top of the calculator GUI, showing the expression being entered.
  + **Valid Inputs**: The display updates with any input from the GUI, reflecting numbers, operators, and any changes to the expression. Valid inputs are any characters that form part of a calculable expression, including parentheses for indicating operation precedence.
  + **Resulting Outputs**: The output is the live updating of the expression as the user enters it and, upon calculation, the display of the computed result.
* **Hyperlinks to Additional Resources**:
  + **Screen Format**: Links are formatted as clickable text or buttons that are clearly distinguishable from the calculator interface, often found at the bottom of the page or in a separate 'Help' or 'Documentation' section.
  + **Valid Inputs**: The user can click or tap on these links to navigate to additional content.
  + **Resulting Outputs**: Clicking a link takes the user to a new page where the additional information or documentation is displayed.
* Data Validation and Error Handling
* **Valid Input Formats**: The system accepts numeric inputs (integers and decimals), arithmetic operators (+, -, \*, /), and parentheses for grouping. Inputs are validated in real-time, with immediate feedback provided for any invalid input.
* **Error Outputs**: When an invalid input is detected, the system provides feedback through error messages displayed on the screen or by highlighting the erroneous part of the expression in the textbox.
* **Currently, the User Interface prototype is local host and we have not been able to deploy it onto a webpage I can attach the link to**

# Quality

* **Extensibility**
* **Modular Design: The system's modular architecture, with clear separation between the Expression Parser, Calculation Engine, and Result Formatter, allows for easy expansion. New features, like additional mathematical functions, can be integrated without disrupting existing functionality.**
* **Service-Oriented: The Calculation Engine is designed as a service that can be called independently, which enables the integration with other systems or migration to a microservices architecture in the future.**
* **Reliability**
* **Error Handling: Robust error handling within the Calculation Engine ensures that unexpected inputs or calculation errors do not crash the system but instead provide meaningful feedback to the user.**
* **Redundancy: Critical components, such as the Calculation Engine, are designed with redundancy in mind to handle high loads and prevent single points of failure.**
* **Portability**
* **Cross-Platform Frameworks: The use of web standards like HTML, CSS, and JavaScript for the User Interface ensures that the webpage is portable across different browsers and devices.**
* **Decoupled Backend: The backend is built with a decoupled architecture, making it independent of the frontend, which allows it to run on various server environments without modification.**