Bonsai Tree

M.A.R.C.O.
Software Architecture Document

Version 1.1

Revision History

Date	Version	Description	Author
06/11/23	1.0	Initial document creation.	Pashia Vang
11/11/23	1.0	Completed Logical View.	Mick Torres
12/11/23	1.0	Revised all sections.	Anya Combs
03/12/23	1.1	Revised class diagrams and package diagrams.	Mick Torres

Table of Contents

Software Architecture Document

1. Introduction

The introduction of this Software Architecture Document (SAD) serves as a gateway, providing a concise overview of the entire document. It encapsulates the purpose, scope, definitions, acronyms, abbreviations, and serves as a roadmap for understanding the M.A.R.C.O. calculator system's architecture.

1.1 Purpose

This Software Architecture Document is a detailed and insightful architectural overview of the M.A.R.C.O. calculator system. It is designed to assist developers, maintainers, and stakeholders by providing detailed insights into the critical design decisions shaping M.A.R.C.O.'s architecture.

1.2 Scope

The scope of this document encompasses the M.A.R.C.O. calculator system, defining architectural boundaries and addressing key aspects such as design decisions, modular components, and interfaces.

1.3 Definitions, Acronyms, and Abbreviations

- SAD: Software Architecture Document
- M.A.R.C.O.: Multi-Arithmetic Righteously Calculated Operations

1.4 References

N/A

1.5 Overview

The SAD contains an overall description of the application that includes the interfaces, requirements, and constraints, as well as descriptions of the application’s functions. Each section of the document is listed below with a short summary.

- Architectural Representation — Describes views representing M.A.R.C.O.'s software architecture for a holistic understanding.
- Architectural Goals and Constraints — Outlines impactful software requirements, objectives, and constraints, like language choice and module functionality.
- Logical View — Describes M.A.R.C.O.'s design model decomposition, emphasizing package hierarchy and layers. Key components include MyInfixCalculator, MyStack, and MyVector.
- Interface Description — Outlines major entity interfaces, including valid inputs and resulting outputs.
- Quality — Explores architecture's contribution to system capabilities like extensibility, reliability, and portability.

2. Architectural Representation

The architectural representation for the infix calculator involves various views that capture different aspects of the system. Software architecture is typically represented through multiple views, each providing a different perspective on the system. Here are the key architectural views for the infix calculator:

1. Module View:

- Description: Represents the system as a set of interconnected modules or components.

- Model Elements:

- Core modules for tokenization, infix to postfix conversion, and postfix evaluation.
- Input/output handling module for user interaction.
- Error handling module for managing exceptions and reporting errors.

2. Component and Connector View:

- Description: Focuses on the major components and their interconnections.
- Model Elements:
  - Components for tokenization, infix to postfix conversion, postfix evaluation, and user interface.
  - Connectors representing data flows and control flows between components.
  - Interface specifications for communication between components.

3. Execution View:

- Description: Illustrates the dynamic aspects of the system, showing how components interact during runtime.
- Model Elements:
  - Sequence diagrams depicting the flow of control during key operations (e.g., expression evaluation).
  - Interaction diagrams showing communication between different components.

4. Data View:

- Description: Focuses on the data structures used by the system.
- Model Elements:
  - Definitions of data structures, such as the custom stack and vector implementations.
  - Data flow diagrams illustrating how data is processed and manipulated throughout the system.

5. Deployment View:

- Description: Describes the physical deployment of the system, including hardware and software components.
- Model Elements:
  - Specification of the target platform and environment where the calculator will run.
  - Deployment diagrams showing the distribution of components on hardware nodes.

6. User Interface View:

- Description: Illustrates the graphical or textual user interface elements.
- Model Elements:
  - Screenshots or mockups of the command-line interface (CLI) used by the calculator.
  - Descriptions of user interaction and feedback.

7. Error Handling View:

- Description: Focuses on how errors are detected, reported, and handled.
- Model Elements:
  - Error handling modules and components.
  - Flowcharts or diagrams depicting the error-handling process.

8. Implementation View:

- Description: Provides insights into the structure of the source code.
- Model Elements:
  - Class diagrams showing the organization of classes and their relationships.
  - Code snippets illustrating key algorithms and functions.

Each view contributes to a comprehensive understanding of the system's architecture. These views collectively capture the structural, behavioral, and deployment aspects of the infix calculator, providing a roadmap for development, maintenance, and future enhancements.

3. Architectural Goals and Constraints

Goals:

- Reusability- we want the program to be able to be used repeatedly without fail.
- Easy to interact with- should be easy to use calculator.
- Quick execution times- the program executes within a reasonable time.
- Good UI.
- Platform compatibility-able to work on multiple OS.

Constraints:

- C++
- All functions must work using the modules we make.
- Must be accessible from the cmd prompt.

4. Logical View

This section will describe architecturally significant parts of the design model. This includes its decomposition into subsystems and packages, and for each package, its own decomposition into classes and class utilities.

4.1 Overview

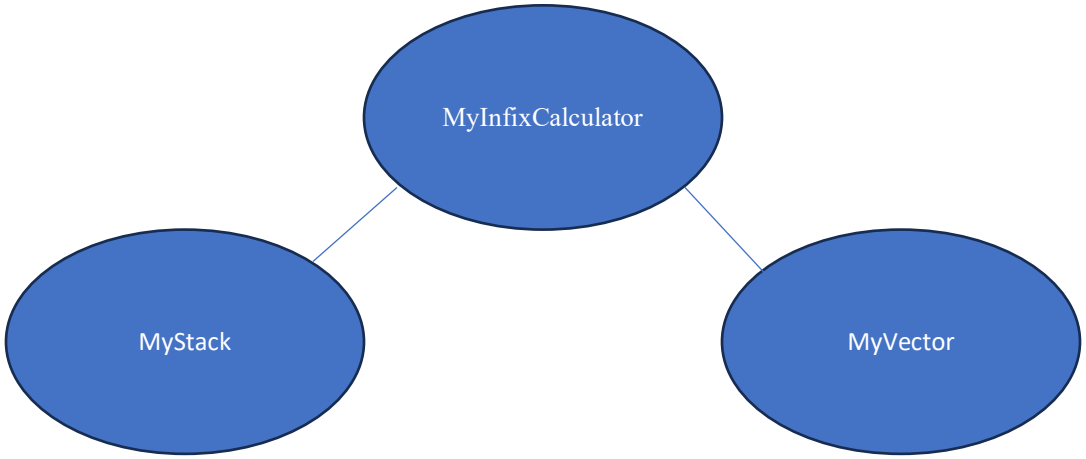
The design model is broken up into one package that is broken up into two classes. All of the classes have multiple class utilities. The package is: MyInfixCalculator. This package is made up of two classes: MyStack and MyVector, and many class utilities. MyStack and MyVector are also made up of many class utilities.

4.2 Architecturally Significant Design Modules or Packages

MyInfixCalculator: This package handles all of the calculations, precedence, and syntax. This means that it will ensure that the equation given by the user is a valid equation. It will check to see if there is a correct amount and arrangement of parentheses and will calculate which parts of the equation to solve first based on the precedence of different operators. It will then calculate the expression to come to an answer. The two classes that are used in this package are MyStack and MyVector.

MyStack: This class creates a stack object, which is a FILO data structure. This is used in the calculator in many ways such as operator precedence.

MyVector: This class creates a vector object which is a sequence container representing arrays that can change their size during runtime. This is used in MyStack and MyInfixCalculator in many ways, but mainly as an array-like object.



MyStack
MyVector<DataType>
MyStack(initSize: size_t)
MyStack(const MyStack& rhs)
MyStack(MyStack&& rhs): MyStack&
~MyStack()
operator=(const MyStack& rhs): MyStack&
operator=(MyStack&& rhs): MyStack&
push(const DataType& x): void
push(DataType&& x): void
pop(): void

©Bonsai Tree, 2023

top(): const DataType&
empty(): bool
size(): size_t
capacity(): size_t

MyVector
size_t theSize
size_t theCapacity
DataType* data
const size_t SPARE_CAPACITY
MyVector(initSize: size_t)
MyStack(const MyStack& rhs)
MyStack(MyStack&& rhs)
~MyStack()
operator=(const MyStack& rhs): MyStack&
operator=(MyStack&& rhs): MyStack&
push(const DataType& x): void
push(DataType&& x): void
pop(): void
top(): const DataType&
empty(): bool
size(): size_t
capacity(): size_t

5. Interface Description

The infix calculator primarily interfaces with users through a command-line interface (CLI). The major entity interfaces include:

1. User Input Interface:

- Screen Formats: The calculator awaits user input in the form of mathematical expressions, supporting integers, decimals, and common operators such as +, -, *, /, %, ^, or. The input follows standard mathematical notation.

- Valid Inputs: Valid inputs include well-formed mathematical expressions adhering to standard notation. The calculator also accepts negative numbers, parentheses for grouping, and optional spaces for readability.

- Resulting Outputs: The calculator outputs the evaluated result of the mathematical expression or an error message in case of invalid input.

2. User Output Interface:

- Screen Formats: The output is displayed on the command-line interface, indicating the result of the mathematical expression or an error message.

- Valid Outputs: Valid outputs include numerical results of evaluated expressions or informative error messages detailing issues such as syntax errors or division by zero.

- Resulting Actions: The calculator waits for additional input or terminates based on user interactions.

3. Error Handling Interface:

- Screen Formats: In case of errors, the calculator outputs error messages on the command line.

- Valid Error Messages: Error messages inform users of issues such as invalid syntax, division by zero, or unrecognized operators. The messages guide users on correcting input.

- Resulting Actions: Users can modify their input based on the provided error messages to obtain a valid result.

4. Extensibility Interface:

- Screen Formats: N/A (Operates behind the scenes)

- Valid Inputs: The architecture supports the addition of new features or operators for future extensibility, allowing developers to enhance the calculator's functionality.

- Resulting Outputs: N/A (Operates behind the scenes)

While no User-Interface Prototype Document is specified, the CLI nature of the calculator simplifies the interface, making it straightforward for users to input mathematical expressions and receive corresponding results or error messages on the command line.

6. Quality

The software architecture of the infix calculator enhances its extensibility, reliability, and portability. Its modular structure supports easy feature expansion without compromising the entire system, fostering adaptability to future requirements.

The reliance on established data structures and robust algorithms contributes to reliability, ensuring graceful handling of unexpected inputs.

The design's focus on platform-independent components promotes portability across diverse environments.

While not explicitly addressing security and privacy, the separation of concerns minimizes vulnerabilities, providing a solid foundation that can be further fortified with additional security measures during implementation.

©Bonsai Tree, 2023