Kung-Fu Programmers

Boolean Expression Evaluator
Software Architecture Document
Version 1.0

Revision History

Date	Version	Description	Author
14/Apr/24	1.0	Initial Publication	Schmidt, S., Stonestreet, B., Rodenberg, S., Medallada, S., Whitmer, K.

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

The Software Architecture Document defines the processes and strategies behind the Boolean Expression Evaluator system's creation. It describes the approach to the design of the software and is the mid-level plan generated and used by managers to direct the construction effort.

The following people use the **Software Development Plan**:

- The Configuration Manager uses it to plan the system's architecture and track construction progress during the scrum.
- **Programming team members** use it to understand how the system will be constructed and expectations for their work.

1.2 Scope

The **Software Architecture Document** defines the bounds of the construction process. It is the directive of the **Configuration Manager** and provides guidance to the programming teams.

1.3 Definitions, Acronyms, and Abbreviations

See the Project Glossary.

1.4 References

For the **Software Architecture Document**, the list of referenced artifacts includes:

- Software Requirements Specification v1.0, 24/Mar/24, KFP
- Glossary v1.0, 25/Feb/24, KFP
- Configuration Management Plan v1.0, 25/Feb/24, KFP
- Project Management Plan v1.0, 25/Feb/24, KFP

1.5 Overview

This **Software Architecture Document** contains the following information:

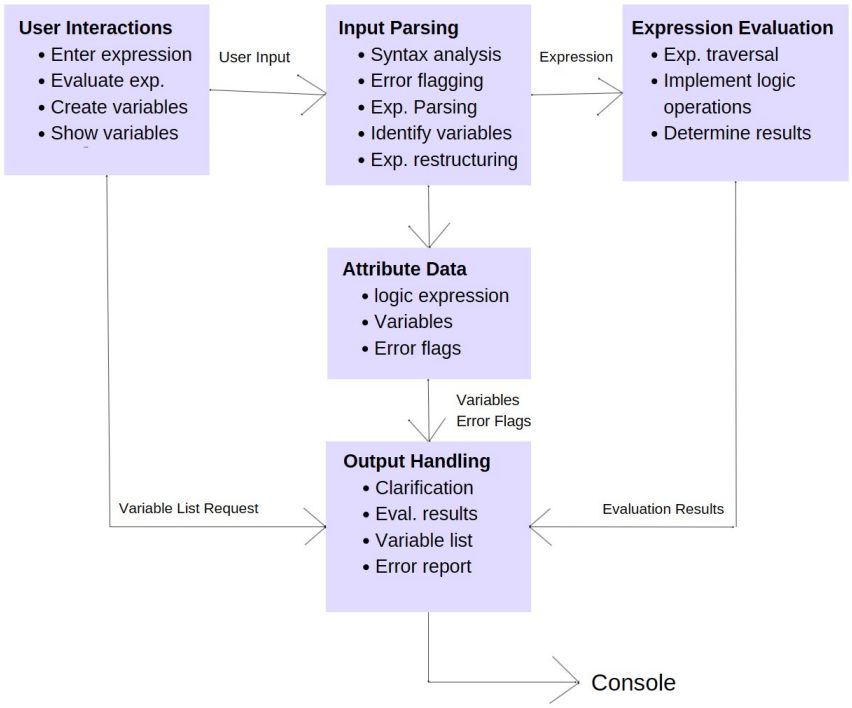
Architectural Representation	—	Describes what software architecture is for the current system, and how it is represented.
Architectural Goals and Constraints	—	Describes the software requirements and objectives that have some significant impact on the architecture.
Logical View	—	Describes the architecturally significant parts of the design model, such as its decomposition into subsystems and packages.
Interface Description	—	Describes the major entity interfaces, including screen formats, valid inputs, and resulting outputs.
Quality	—	Describes how the software architecture contributes to all capabilities (other than functionality) of the system.

2. Architectural Representation

The following architectural modeling views represent the Boolean Expression Evaluator in terms of multiple perspectives to address all stakeholder concerns. Each view illustrates a specific set of system characteristics, described by corresponding key elements and their purposes within the system.

System Processes:

An overview for the BXE system's processes focused on the dynamics of its runtime behavior.



The above diagram outlines the sequential flow of data processing throughout interaction of the system's user and processing components.

Logical Overview:

This is a descriptive list representation of the BXE's primary functionality offered to the end user.

- A user-friendly, user interface for system navigation and operation selection
- A robust user input evaluation and error detection process
- A central structure to store, manage, and access operational data throughout system processes
- An output handler for error alerts, expression evaluation results, and user-request responses

Developmental View:

A representation of the BXE software functionality in terms of the primary packages and data structures that model the logical operations and mechanics of expression evaluation.

System-User Interaction:
- UI accessing displays a selection menu of these operations:
  - Enter a Boolean expression to be evaluated
  - Enter a new variable and assignment value
  - Request current variables to be displayed

Input Processing:
- The handling of user prompted input for expressions and variables includes:
  - Syntax analysis and error flagging
  - Variable identification and processing
  - Expression parsing and restructuring for evaluation

Data Storage:
- A central data structure stores the following attributes for each instance:
  - User-expressions
  - Variable list
  - Error flags

Expression Evaluation:
- Boolean expressions without errors are evaluated by way of:
  - Expression traversal
  - Implementation of logic operations

System Output:
- To ensure reliably efficient user experiences, the system's output process constructs string-representations of the following:
  - Expressions evaluation results
  - Current defined variables
  - Error reports

3. Architectural Goals and Constraints

- Safety
  - Safety of programming teams is paramount and the only concern of this design.
  - This project is minimal risk, but care will be taken to ensure that team members feel empowered to report any mental health requirements during the construction process.
  - Any feelings of numbness/tingling in the hands during any "sprints" will need to be reported.
- Security
  - There will be no user PII (Personally Identifiable Information) or other data that requires consideration of a security framework.
- Constraints
  - This program will be written in C++.
  - This program must be compiled and executed on the KU EECS Cycle Servers.
  - See the Project Management Plan and the Software Requirements Specification for more details.

- Development tools
    - Programming: Visual Studio Code, Notepad++, etc.
    - Software Management: GitHub
    - See the Configuration Management Plan for more details.
- Schedule
    - Week 1: 4/15-21
        - Expression Struct. Package
        - Parser Package
        - Evaluator Package
    - Week 2: 4/22-28
        - UI Package
        - Output Package
- Legacy Code
    - N/A
- Design and Implementation Strategy
    - Design:
        - Front-end UI layer with Pipe-and-Filter back-end execution
    - Implementation:
        - Scrum: 2 1-week “sprints”

4. Logical View

4.1 Overview

Packages included in this program

- Expression struct package
- UI package -> Parser Package
- Parser package -> Evaluator Package
- Evaluator package -> Output package
- Output package - > UI package

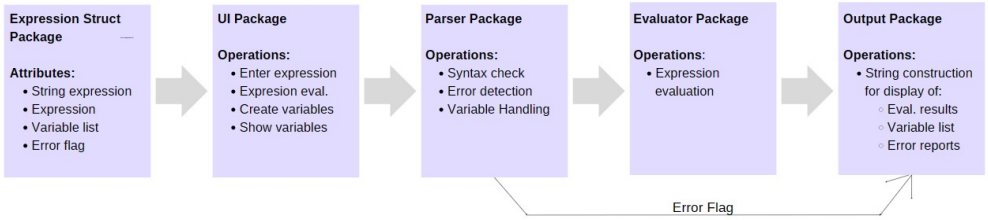
4.2 Architecturally Significant Design Modules or Packages

- Expression Struct package
    - This package includes A string expression, an int flag, an expression, and a variable list. In the string expression is where the Boolean expression to be solved is. The int flag is changed to a number representing a certain error if there is any found during the process. The list is used to store variables to be used during the evaluation process. The output string is where the final output will be stored. This string will be written by the output package.
- UI package

  - This package will include everything the user can interact with and will return to the user the evaluated expressions. Once the user enters his input the expression struct will then be sent to the Parser package.

- Parser Package
  - This package parses the expression struct and validates it before it is sent to the evaluator. While parsing, it checks for syntax errors and ensures all variables used in the program are defined. If there are any problems, it will flag the error in the expression structure. Then it will replace any variables in the expression with its respective value. Once done it will send the expression struct to the evaluator package.
- Evaluator package
  - This package takes the expression struct and solves it completely. It does not need to worry about errors as the parser has already checked for them. But if there is an error flag risen by the parser, the Evaluator will skip all of its steps and send the expression struct to the output as the expression will not be able to be solved.
- Output package
  - This package takes the expression struct and prepares a string for the UI package to return to the user. Based on what the expression structs contents are it will return different strings to the user. If there is an error flag, the string returned will be the respective error associated with the error flag. If there are no error flags, then the string returned will be the solved expression.

Architectural Packages Diagram



5. Interface Description

UI

The UI for this program is a simple command line interface, where the user interacts with the program by entering logical expressions. The program will then either calculate the problem and return the truth value or will return an error with what is wrong in the expression.

Output

The program's output is a text line which will display the truth value or error. The truth value of the equation will be given by printing either true or false in the output line. If the program detects an error, it will return that information in the output line as well as giving a short description of the problem it detected in the equation.

UI and Output

```
== RESTART: C:/Users/Owner/OneDrive/Desktop/EECS 348/Project/Example output.py =
======Boolean Logic Simulator======


Enter Logical Expression: (T|F)$F
Evaluation: True


Enter Logical Expression: !(T&T)
Evaluation: False


Enter Logical Expression: !&T
ERROR: Missing operand, no operand after NOT


Enter Logical Expression:
ERROR: No expression, no operands or operators in command line


Enter Logical Expression:
>>>
```

(Not final UI or output design)

Valid Input Constraints

- Missing Operand
  - Valid inputs should not have a missing operand after a logical operator.
- Unknown Operator
  - Valid inputs should not contain unrecognized operator symbols.
- Mismatched Parentheses
  - Parentheses should be properly matched in expressions.
- Circular Logic
  - Expressions should not define a variable in terms of themselves.
- Empty Expression
  - Input expressions should contain both operands and operators.
- Double Operator
  - Consecutive AND or OR operators are not allowed.
- Missing Truth Values
  - All variables in expressions should have assigned truth values.
- Inconsistent Characters
  - Use consistent characters for representing truth values.
- Operator and Operand
  - Operators should not appear immediately after operands without an intervening logical operator.
- Invalid Characters
  - Only use uppercase letters "T" and "F" to represent truth values.

6. Quality

Reliability: high cohesion and reduces coupling

- We can use high cohesion through each logical operation (AND, OR, NOT, NAND, XOR) in a separate filter.
- Each filter will perform a logical operation.
- Connecting the filter through pipes, it'll make the data flow smoothly from one filter to another and reduce the coupling.
- Each filter will work on its own independently.

Portability: the programs are compiled on any device. Relaxed constraints increased portability.

- Compile and run the pipe and filter through the C++ compiler.
- It would make the logical operators run through the simulator while increasing portability.

Extensibility: high modularity

- This would extend the code by adding new filters for the features or logical operations from the pipe and filter.
- If the author would like to add new features, it would enhance the whole project.
- They can add any logical operations if they would like to.