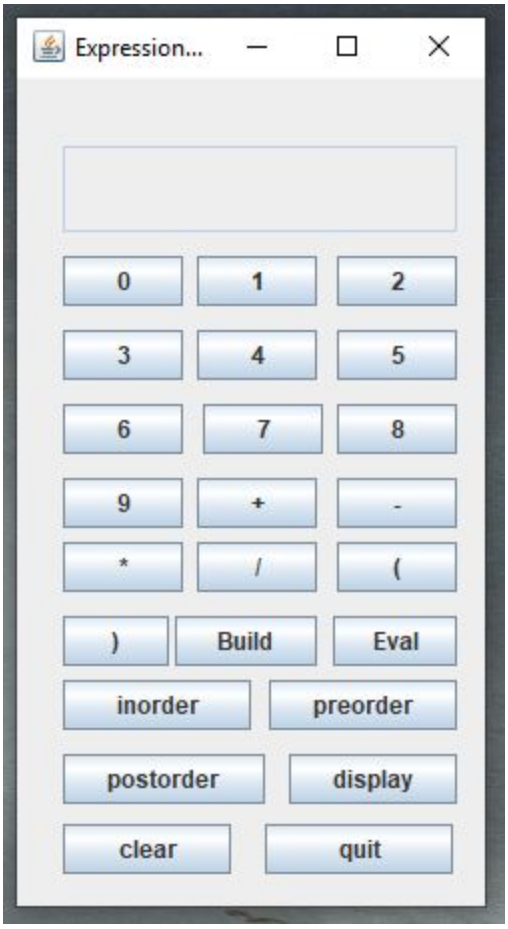
This document briefly explains the Java app to evaluate an infix expression. The goal of the project is to implement at least 12 design patterns and one architectural pattern to get the working version of the app.
The evaluation requires the following steps:

- Parsing the expressions into tokens
- Creating a parse tree using evaluation grammar
- Creating an abstract syntax tree using the parse tree
- Evaluate the expression using infix evaluation order
- Print tree in a level order
- Print tree traversals i.e. [In|Pre|Post]order

The following sections will briefly describe the various patterns and corresponding classes. The complete class diagram is attached along with this report. The class diagram is too big to be part of this document. The following picture is used as a reference for explaining various patterns.



Valid Input - Single digits, operators, & fully parenthesized expressions. ((1+2)*(5-7))

Invalid input - (11+2+3), 1+2+3 etc

Diagrams

Class diagrm depicted entire application is attached.

Sequence diagrams for the following scenarios are attached

- **Build Tree**
- **Inorder Traversal**
- **Levelwise print with Chain of responsibility**
- **Infix evaluation using visitor**

Please refer the diagrams to relate the text below. I have uploaded both the jpeg images and rational project for the diagrams. The sequence diagrams are given numbering to follow the order for better understanding.

1. **Command**

   Following buttons are implemented using the command pattern
   Build - builds an expression tree from the input expression
   Eval - evaluates the expression tree and shows the result
   [in|pre|post]order - Prints inorder traversal for the tree built using build button
   Display - Levelwise tree printing on console
   Clear - erases the tree and expression from memory and display
   Quit - terminates the program

   Abstract class : UserCommand
   Concrete commands - BuildCommand, ClearCommand, DisplayCommand, ClearCommand

2. **State**

   Abstract class - State
   Concrete States - InitializedState, UnIntializedState
   TreeOps - Context class
   The app toggles between two concrete states i.e. InitializedState, UnIntializedState
   Starts state - UnIntializedState
   After build command state becomes IntializedState
   The clear command brings is back to UnIntializedState.
   The display/[in|pre|post]order/eval works only in IntializedState

3. **Composite**

   The tree is formed using a composite pattern.
   Node - Component class
   Operator - Composite class
   Leaf - Leaf class

4. **Bridge**
   A separate hierarchy is created for representing an expression tree. Following classes define bridge.
   Abstract class (Left side) - ExpressionTree
   Concrete tree (Left side) - logOpsExpressionTree (Logs the function calls)
   Abstract class (Right side) - Component i.e. Node class
   Concrete classes (Right side) - Operator and Leaf.

   This separates interface from implementations, i.e. both can vary independently.
   The ExpressionTree holds the root node of the tree.

5. **Iterator**
   The expression tree is traversed using iterators (non-recursive).
   Abstract class - Iterator (next, hasNext methods only)
   Concrete iterators - inOrder, preOrder, postOrder
6. **Factory Method**
   IteratorFactory/UserCommandFactory - A hashmap is maintained to cache the pre-created objects. A factory method returns an appropriate object of choice.
   IteratorFactory is part of ExpressionTree.
   UserCommandFactory is part of TreeOps class explained later.
7. **Decorator**
   nodeDecorator - Abstract wrapper for Node class
   quoteNode - Concrete decorator for Node.
   It is used only for the root node to show the functionality. The decorated root can be seen in the tree traversals.
8. **Visitor**
   Abstract visitor - Evaluator class
   Concrete visitor - infixEvaluator
   ExpressionTree class holds a reference to infixEvaluator. Each of the nodes in the composite hierarchy implements ‘accept’ method and take infixEvaluator as an argument. The ExpressionTree also implements ‘accept’ method which calls ‘accept’ for root.
9. **Chain of responsibility**
   The logger is implemented as a chain of responsibility.
   Logger - Abstract class
   ConsoleLogger - Concrete logger to print a message on a console.
   FileLogger - Concrete logger to print a message in the file.
   TimeStampLogger - Concrete logger to print a message prepended with elapsed time i.e. the time since the user started the program.
   The chain used in the program is *TimeStampLogger-->ConsoleLogger*

All messages in logOpsExpressionTree method are printed using TimeStampLogger. However, level-wise printing the tree is done without time stamps i.e. ConsoleLogger
FileLogger is unused, however, can be added easily to chain for having additional use-case.

10. **Prototype**
    Timestamps are created for each print message using the clone method of ElapsedTime class. The concrete class here is ElapsedTimeMilliSeconds.
11. **The Flyweight**
    The Operator and Leaf classes maintain a glyf variable. It is a string instead of an actual picture. The flyWeight is used to lever-wise print the tree. It would require actual images if done graphically. However, I am only showing the flyweight implementation using console. The strings are maintained in the constant pool, eliminating the need to explicitly cache the glyfs.
    A display method for uses glyf variable to print the node. Glyf is an intrinsic state here, while the level is an extrinsic state.

12. **Singleton**
    A ModelDb class is implemented as a singleton, restricting the instances to one only. It is part of an MVC pattern. The class stores input expression and various traversals. It is used by fronEnd (i.e. View) and TreeOps (Controller) both.
13. **Observer**
    The view consists of many buttons, which are JButton objects. Each of these objects registers by calling a method *addActionListener* . The observer, in this case, is built in Java class *ActionListener*. Each of the buttons acts as subjects.

14. **Template Method**
    The buildTree method acts as a template method. The function calls made inside buildTree are all implemented in concrete tree i.e. logOpsExpressionTree
    The same method can be reused if a different tree implementation is provided e.g. Synchronized tree where all treeOps are guarded by locks.

15. **Layered architecture**
    The application is divided between three layers as follows.
    View - frontEnd class
    Controller - treeOps (Also a facade for all the complex tree handling.)
    Model - Singleton DB instance (in-memory persistent variables)

16. **Containerization as an architectural pattern**
    The Dockerfile creates a docker image good enough to run the java app.
    Use build.sh to create a docker image.
    Use run.sh to run the app.
    More details in README (uploaded).

Key Algorithms -

Inorder -

- Push left subtree to the stack
- Repeat until the stack is empty
    - Pop and print the node
    - Push entire left subtree of right child for the popped node

Preorder -

- Print current node
- Push right child of the current node to stack
- Next = current->left
- If current->left == Null
    - Next = pop from stack
- Else
    - Next = current->left

Postorder -

- Create two stacks s1/s2
- Push root to s1
- Repeat until s1 is empty
    - Push left and right of the current child to s1
    - Push current to s2
- S2 has post-order traversal (keep popping)

TODO

- Interpreter pattern to the parser
- Builder to create a composite hierarchy
- Get command line input and apply a strategy pattern to choose between cli or gui
-