Performance Comparison of Design Patterns

Akshay Virkud

April 24, 2015

Abstract

A good software design is the one that makes the software robust, reusable, maintainable, extensible and which helps to optimize its performance. The main goal of this assignment is to analyze the four different versions of expression evaluator and determine which of them performs better. The criterion for performance analysis is the execution time taken by each of the versions for different types and number of expressions. The four versions of expression evaluator that we considered for evaluation are:

1. Expression evaluator using Postfix expression approach.
2. Expression evaluator using Postfix and Flyweight pattern.
3. Expression evaluator using Postfix, Flyweight and Fast Path performance patterns.
4. Expression evaluator using Expression Tree approach.

Each version of the expression evaluator makes use of certain software design patterns. Furthermore, the postfix version of the expression evaluator has been re-factored for optimizing the performance by using Flyweight pattern and software performance patterns such as fast path pattern.

Introduction

The main aim of the expression evaluator is to handle the following operators or tokens: +, -, /, *, %, (,), integers (positive and negative) and follow operator precedence. All the inputs to the program came from the STDIN and all the outputs were displayed using STDOUT. Initially we started with the basic array version of the expression evaluator and over the coursework we came up with two major forms of expression evaluator – postfix expression evaluator and the tree based expression evaluator. In this assignment in order to analyze the performance we prepared two more variants of the postfix version – postfix with Flyweight and postfix with Flyweight and fast path pattern. Following is the brief description of each of the expression evaluator that we have used in this assignment.

Postfix Expression Evaluator: This expression evaluator was built up on the stack based expression evaluator. In this approach we first converted the infix expression to post expression using the Infix to Postfix algorithm. Then we use the *Command Pattern* to evaluate the Postfix expression, and the *(Abstract) Factory* to create the commands based on parsing the Infix expression to convert it to a Postfix expression.

Postfix Expression Evaluator with Flyweight Pattern: The purpose of this variant was to optimize the performance of the postfix expression tree evaluator by using the Flyweight pattern. *Flyweight* pattern is used when the application has a large number of objects and the storage costs are high because of the large quantity of objects. The main intent of this pattern is to use sharing to support large numbers of fine-grained objects efficiently. In our case by using flyweight objects we reduced the number of operator objects that used to be created in our application. This is possible because the operator objects are stateless. However number command objects have state and hence they cannot be reused.

Postfix Expression Evaluator with Flyweight Pattern and Fast Path performance pattern: This variant of expression evaluator is built on the Flyweight postfix version. Here in addition to Flyweight pattern we have also introduce a *Fast Path* performance pattern in the expression evaluator. The purpose of the fast path pattern is to improve the response time by reducing the amount of processing required for dominant

payloads. Thus it handles the most commonly occurring tasks more efficiently than the 'normal path'. In our case we have assumed that the evaluation of addition operator is the most commonly occurring task and hence we have re-factored our flyweight postfix design to optimize the performance of addition operation. In order to test this approach we created a suite of input files (type4) which contain a majority of add operations.

Tree Based Evaluator: In this approach instead of converting the infix expression to postfix expression, we directly converted the infix expression to tree which is a *Composite* Object. We made use of the *Abstract Factory* pattern to create the tree nodes and then the *Builder* pattern for creating tree. We then implemented an eval() method on the Composite to evaluate the expression tree, and also the *Visitor* pattern to evaluate the expression tree.

Methods

The touchstone for measuring performance is the time required to execute a set of input expressions. As a result we prepared several input files which varied in terms of type of expressions and number of expressions. In order to generate these input files, I created a stand-alone application in C++ which took the number of expressions, filename and the type of expression to be generated as input from the user and generated a .txt file containing input expressions as output. I created following four types of expressions:

a. number operator number
b. (number operator number) operator (number operator number)
c. number operator number operator number
d. number + number + number + number + number + number

Where *number* is any random number generated from 1 to 99 that was generated using *rand ()* function which returns a pseudo-random integral number and *operator* is any one of the following operators: +, - ,*, / and %.

For each expression type we created three versions of files:

i. File with 500 expressions
ii. File with 1100 expressions
iii. File with 2100 expressions

These input files were then given to the expression evaluator program as input.

The expression evaluator program makes use of the *Strategy Pattern* for selecting the type of expression evaluation strategy that needs to be used. The user enters the expression evaluator type and the input file path along with the file name and file extension and then the program evaluates the input expressions and records the execution start timestamp and execution stop timestamp. For measuring time we made use of the C++ *clock ()* function and the total execution time in milliseconds is calculated using the formula:

*Total Execution Time = 1000 * (stop - start) / (double) CLOCKS_PER_SEC;*

The results that we obtained from our overall analysis are discussed in the next section.

We also generated Valgrind reports for all the expression types that we tested in order to detect memory leaks.

Results & Discussions

Figure 1: Graph describing the execution time required for Expression Type 1.

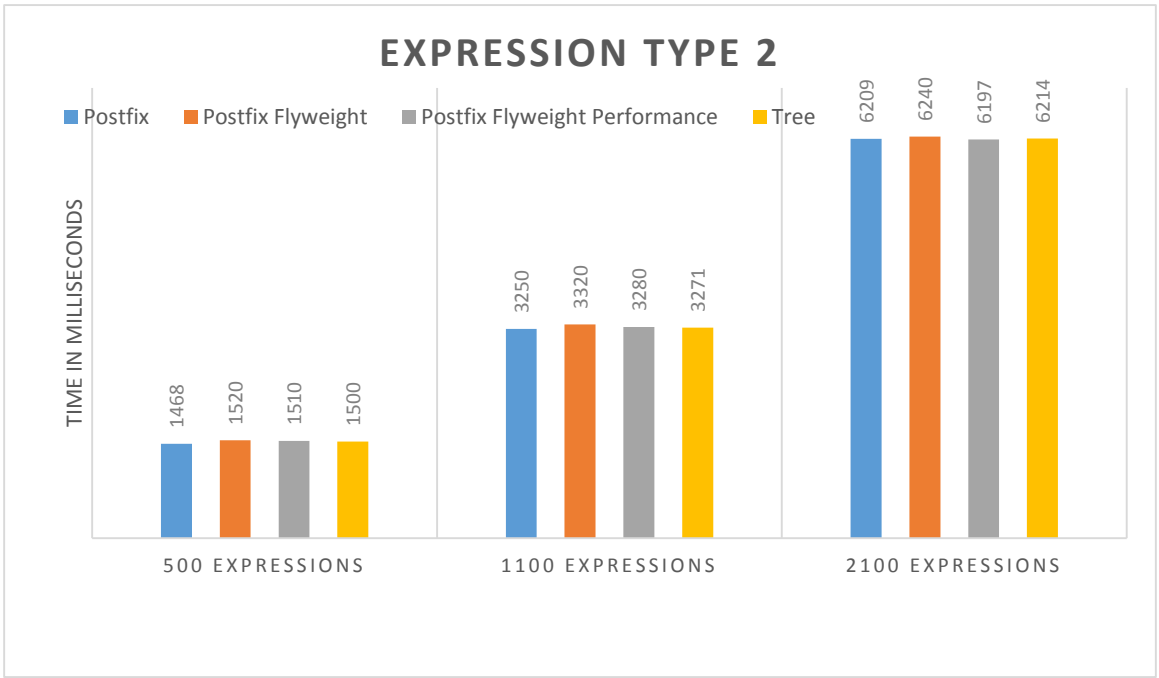

Figure 2: Graph describing the execution time required for Expression Type 2.

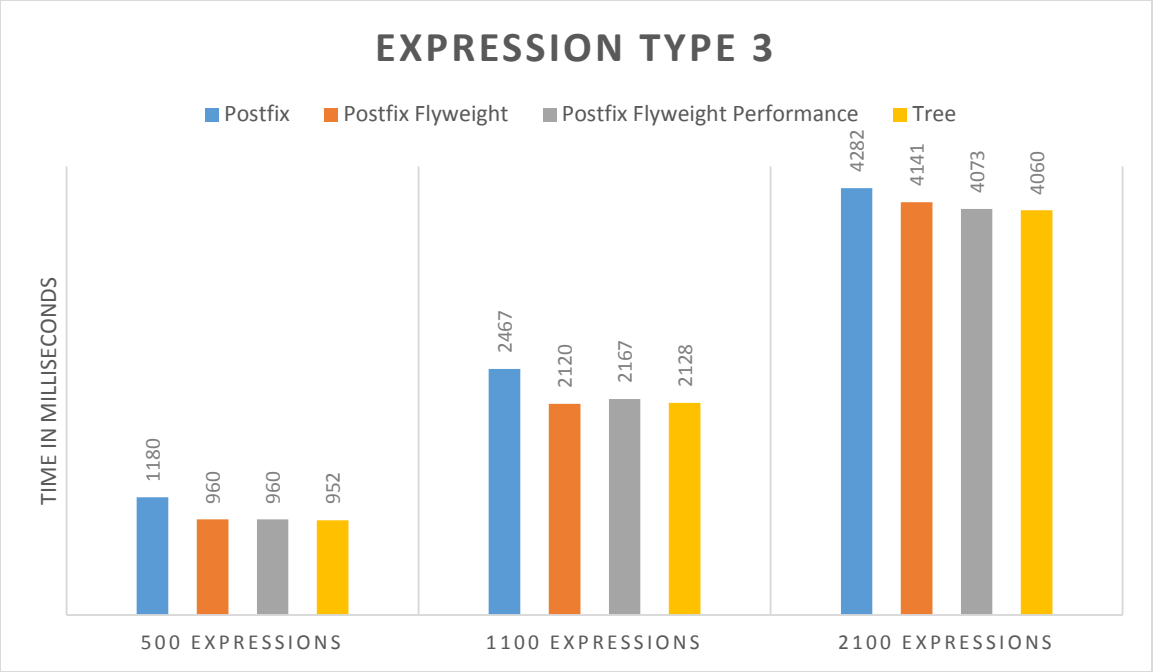

Figure 3: Graph describing the execution time required for Expression Type 3.

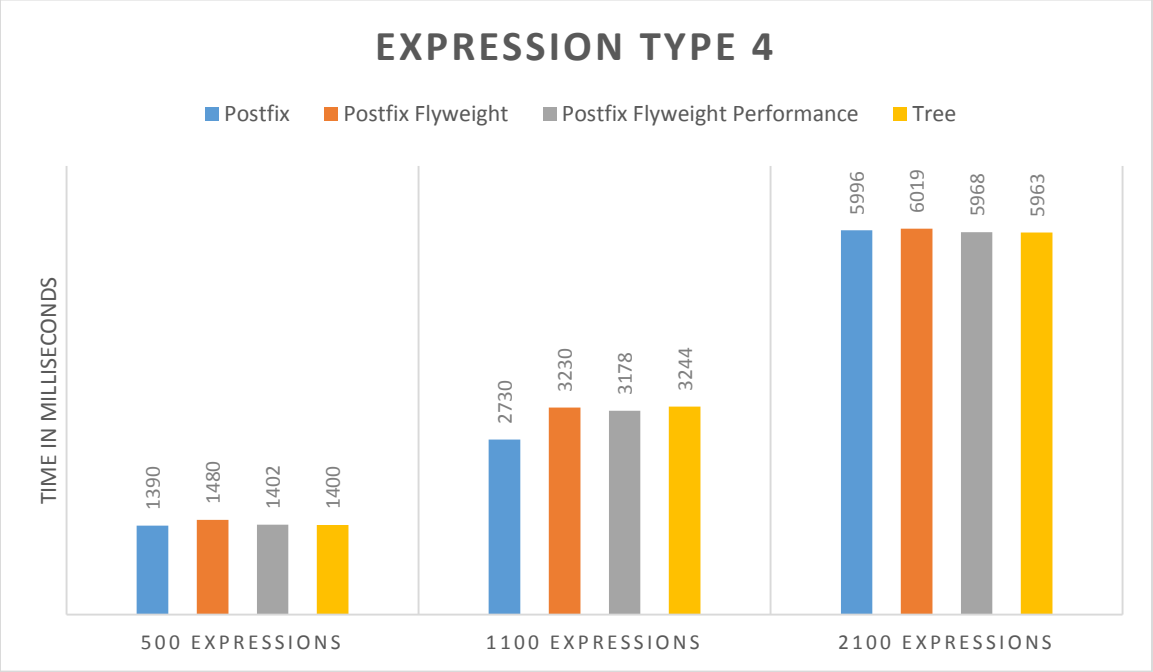

Figure 4: Graph describing the execution time required for Expression Type 4.

Following are the results of our tests:

- For the type 1 expressions (*number operator number*) we found that the time taken by the expression tree version was the least even when we tested the expression evaluators for input sizes of 500, 1100 and 2100 expressions respectively. On the other hand the postfix expression evaluator took the largest amount of time for execution. For input file containing 500 expressions, the time taken by postfix evaluator was almost 1.5 times the time taken by the expression tree version.

- On increasing the expression size a bit i.e. for type 3 expressions (*number operator number operator number*) the behavior remained same- the tree version, postfix with Flyweight version and postfix with Flyweight and performance pattern were faster as compared to the postfix version. This behavior did not change even when we increase the number of expressions per file from 500 to 2100.

- For more complex expressions like type 2 expressions (*(number operator number) operator (number operator number)*) which had parenthesis in them, the behavior changed. When we tested our expression evaluators for type 2 expressions, we found that postfix expression evaluator performs better as compared to the postfix with Flyweight.

- The type 4 expressions (*number + number + number + number + number + number*) were specially designed for testing the postfix with Flyweight and performance pattern and we found that as compared to the postfix with Flyweight, the postfix with Flyweight and performance pattern was faster by about 70 – 80 milliseconds in all the three input files (i.e. 500 expressions, 1100 expressions and 2100 expressions).

Below is the screenshot of the system specifications of the machine that was used for testing:

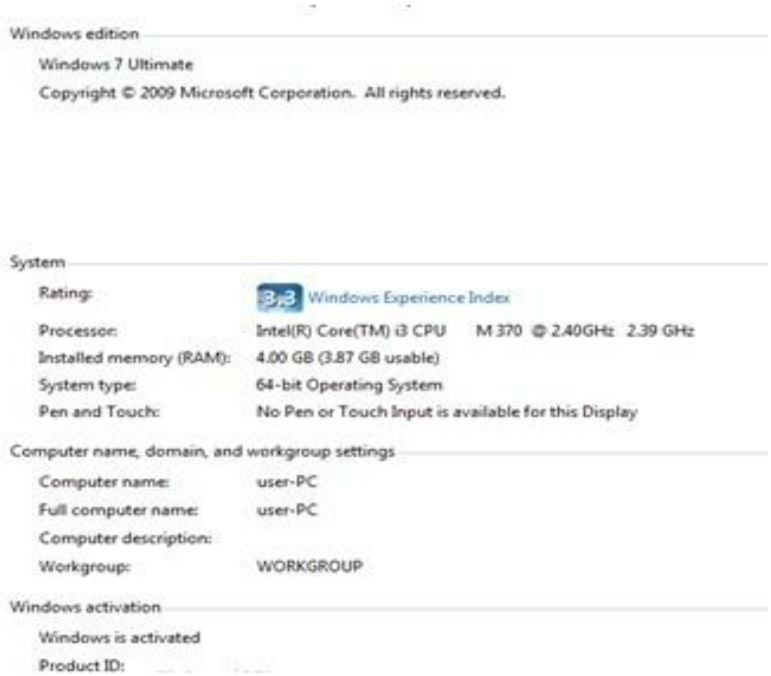

Figure 5: System Specifications

Concluding Remarks

From the above obtained results, we can say that all the versions of expression evaluator have their own pros and cons and there is no particular version that always performs better than the others. All the forms of expression evaluators that we considered have been implemented using good software design patterns and hence each one has its own advantages.

In general the expression tree version of the expression evaluator has a better performance.

However in the type2 expressions the performance of postfix is better than the postfix with Flyweight and postfix with Flyweight and Fast Path performance pattern because in case of Flyweight pattern if the expression contains all different operators then there is unnecessary time consumption in verifying whether the copy of operator exists or not. This makes the Flyweight pattern slower.

In the type4 expressions, majority of the operators were addition operator. We had one version of our postfix optimized for addition operation and we found that the postfix with Flyweight and Fast Path performance pattern performed better than the postfix with only Flyweight as it had to pass through less conditional statements.

Besides that the result of the Valgrind reports show that all the memory that was being used by the expression evaluators was finally properly de-allocated and there was no memory leak for any of the versions.

References

1. Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley Professional, 1994 , ISBN 0-201-63361-2.
2. SoftwareDesignPatterns, Software Performance Patterns and Flyweight course slides by Dr. James Hill for CS 50700: Object-Oriented Design and Programming course.
3. http://en.wikipedia.org/wiki/Fast_path