CS 251: Intermediate Software Design

Program Assignment 4

This programming assignment focuses upon using a variety of patterns to implement a non-trivial program. The final result will be several thousand lines of code long, though you'll reuse all the earlier `Array`, `AQueue`, and `LQueue` classes, so it won't seem as painful as writing/debugging thousands of lines of code from scratch! The assignment is split into the following three parts so you can separate concerns and create/debug your solution incrementally.

Part A. We'll start by implementing using the Adapter pattern, which you'll use to integrate your `LQueue` and `AQueue` into a new `Queue` and `Queue_Adapter` template class hierarchy that can be used to dynamically select which type of queueing strategy to use in the program at runtime. The use of Adapter ensures that no changes are required to the existing `LQueue` and `AQueue` classes. You'll also need to implement an adapter for the STL `queue` class so that it can be plugged into the `Queue_Adapter`, as well.

You can get the shells for Part A of the program from `www.dre.vanderbilt.edu/~schmidt/cs251/assignment4a`. You'll need to reuse the files from your `Array`, `AQueue`, and `LQueue` implementations.

Part B. In this part of the assignment you'll use the `Queue` and `Queue_Adapter` template classes to implement a program that will build and traverse a binary tree using various traversal strategies. You'll use the following patterns to guide the design of Part B:

- **Singleton**, which is used to implement an `Options` singleton that parses and keeps track of the command-line options.
- **Factory**, which is used to create the appropriate types of queueing and traversal strategies indicated by the `Options` Singleton.
- **Strategy**, which is used to implement the appropriate type of queueing strategy (such as `AQueue` or `LQueue`) and traversal strategy (such as level-order, in-order, pre-order, and post-order).

You can get the shells for Part B of the program from `www.dre.vanderbilt.edu/~schmidt/cs251/assignment4b`. You'll need to reuse the files from your solution to Part A of the program.

Extra graduate student work. Graduate students need to implement the following additional patterns for Part B.

- **Abstract Factory** and **Factory Method**, which are used instead of individual `Factory` functions to consolidate all the factories into a single concrete factory class.
- **Bridge**, which is used to avoid exposing "naked" pointers and to simplify memory management, e.g., by reference counting throughout the program.

The use of these patterns are optional for undergraduates.

Part C. The third part of the assignment will replace the Strategy pattern with the following patterns:

- **Iterator**, which is used to retrieve each element in the binary tree one item at a time, using various traversal orders, e.g., in-order, pre-order, post-order, and level-order.
- **Visitor**, which is a generalization of Strategy used to perform an operation on each node that is visited.

The visitor implementations need only print the contents of the tree when visited, though again your *design* should be capable of being extended to handle other types of visitors to prepare for Part D. Likewise, your visitor implementations only need to define an iterator for level-order traversal, though your design should be capable of being extended to handle the other traversal orders, as well, for Part D.

You can get the shells for PartC of the program from `www.dre.vanderbilt.edu/~schmidt/cs251/assignment4c`. You'll need to reuse many files from your solution to Part B of the program.

Extra graduate student work. Graduate students need to implement the Abstract Factory, Factory Method, and Bridge patterns you used for Part B. Moreover, graduate students need to implement STL-style iterators, whereas these are optional for the undergraduates (who can use GoF-style iterators if they choose).

Part D. This final part of the assignment will change the tree factories to produce an expression tree, which consists of nodes containing operators (*e.g.*, +, -, *, and /) and operands (*e.g.*, integers). You will need to implement the following pattern for the expression tree:

- **Composite**, which treats individual objects (operands) and recursively-composed objects (operators) uniformly.

Likewise, a new visitor will be needed to evaluate the expression tree to print its value. You will therefore need to enhance your iterator to support (at least) pre-order traversal of the tree.

Extra graduate student work. Graduate students will need to enhance the expression tree program to use the Interpreter and Builder patterns to parse expressions input from users to create expression trees.