CS 251: Intermediate Software Design

Program Assignment 4
Part A Due Monday, March 17^{th}, 2008
Part B Due Wednesday, March 26^{th}, 2008
Part C Due Wednesday, April 9^{th}, 2008
Part D Due Wednesday, April 16^{th}, 2008

This programming assignment focuses upon using a variety of patterns to implement a non-trivial program. The final result will likely 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 can get the "shell" for Part A of the program from `www.cs.wustl.edu/~schmidt/cs251/assignment4a`. There are several files that are partially filled out for you. Note that you'll need to reuse the files from your `AQueue`, `Array`, 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 "shell" for Part B of the program from `www.cs.wustl.edu/~schmidt/cs251/assignment4b`. There are several files that are partially filled out for you, but you'll need to do the bulk of the work.

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.

There are no shells for this program - you are free to design your program any way you'd like, as long as there are no memory leaks and the design/implementation is clean.

Extra graduate student work. Graduate students need to implement the Abstract Factory, Factory Method, and Bridge patterns you used for Part B.

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). 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) in order traversal of the tree.

Extra graduate student work. Grad students need to implement the following pattern for the expression tree:

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

The use of this pattern is optional for undergraduates.