Review 4

CS202 Programming Systems, Summer 2020

Armant Touche

Date: August, 22, 2020

Design Analysis

For this assignment, we were to re-implement a food ordering system in Java via an Integrated Development Environment (IDE) application to more quickly learn Java. Program five was to create AVL tree of orderers where each node is Linear Linked list of orders created by that orderer. For the Object Oriented Design, I simply create parent for the Order class, which encapsulates the orderers in the AVL tree. This data encapsulation makes managing the AVL tree much easier and especially when doing the right-rotation and left-rotation during insertion into the AVL tree. Food acted as the "glue" for my derived classes which possesses two virtual function called read-in and display. I could have done the same with getter's and setter's but I just left those methods in the Food class because of simplicity. Since the data structure for this program was a linear linked list, I had both a previous and next self-referencing pointers and the corresponding getter's and setter's that deal with both of the previously mentioned pointers. The getter's and setter's both return references of Food-type. For each of the derived classes, they had unique data fields and method but all share the data members in the base class. The methods weren't too hard to implement in small slices and using the IDE really streamlined the developing process significantly. For the linear linked list manager, I had a class called Order which has a containing relationship with the base class Food. For the Order class and all the classes including Main were derived from a class called Util which is short for utility. The utility class abstracted the Scanner method and some of prompts which prevented redundant code from being implemented. The Order class had the methods the client would be able to implement. The methods were create-order, display-all, display-orderer, destroy-all, and delete-order. These methods can and were called from the client program which allowed the user to use the food ordering system we were required to implement. In the create-order method, the linear linked list was partly managed from here. If no orders were entered previously, then a new order was created with the make-item method present in Order's method field. The make-item method returned a Food reference and that reference was read-in in the create-order method. If order already exists, then I pass the reference created by make-item into the recursive-insert function. The recursive-insert method is implemented with tail-recursion, checking if the current item orderer's name matches the name of the creator. If the name of creator doesn't match, then traverse to the appropriate alphabetical location in the linear linked list. The insertion into the linear linked list happens alphabetically and when a match occurs, the order is inserted after the match. I could've done before but I would have to account for three cases in the wrapper method (create-order). I do not keep track of the number of orders. For the display-all function, I have both a wrapper and recursively display-all method. The recursive display-all method uses tail-recursion for displaying all the existing order in the linear linked list. The base case for this recursive method is that the user traversed to the end of the list. For delete-order method, the user supplies the name of the orderer they want to delete from. The deletion can happen more than once, as long as their multiple orders under the same name of user-supplied name. I allow the user to pick and choose which order to delete but I allow them in a forward, sequential manner. Cannot go back to a previous order. If they wanted to go back, then they would have to go back to the client program and pick the same option again.

Debugger

Geeksforgeek.com and Intellji Idea IDE was perhaps the biggest time saver because working with an AVL tree data structure was a challenging task. Stepping through Intellji Idea IDE really solidified how Java worked and was especially important when tracing insertion and extraction operator in both of the derived classes that needed to be implemented. Another debugging program that aided me was Idea debugger, which helped with tracing insertion. I used watch, breakpoints, and the invocation trail a lot to aide in debugging program four. Working the base class and it's virtual methods in Intellji Idea IDE really helped solve issues like reading incorrectly or displaying data members from the base class and not any of the derived one. I realize a little to late into programming that data structure for program was JUST a linear linked list and not a binary search tree of linear linked, which I had to refactor my a little bit. Thankfully I already implemented the DLL manager. I had to pull an audible a little late in the game but I think I am pretty set up for program number two. I just need to learn return type identification and abstract base class concepts which will be covered next week. I will there was a straightforward way to access data members in a derived class via dynamic binding but I need to practice anyway. No worries.