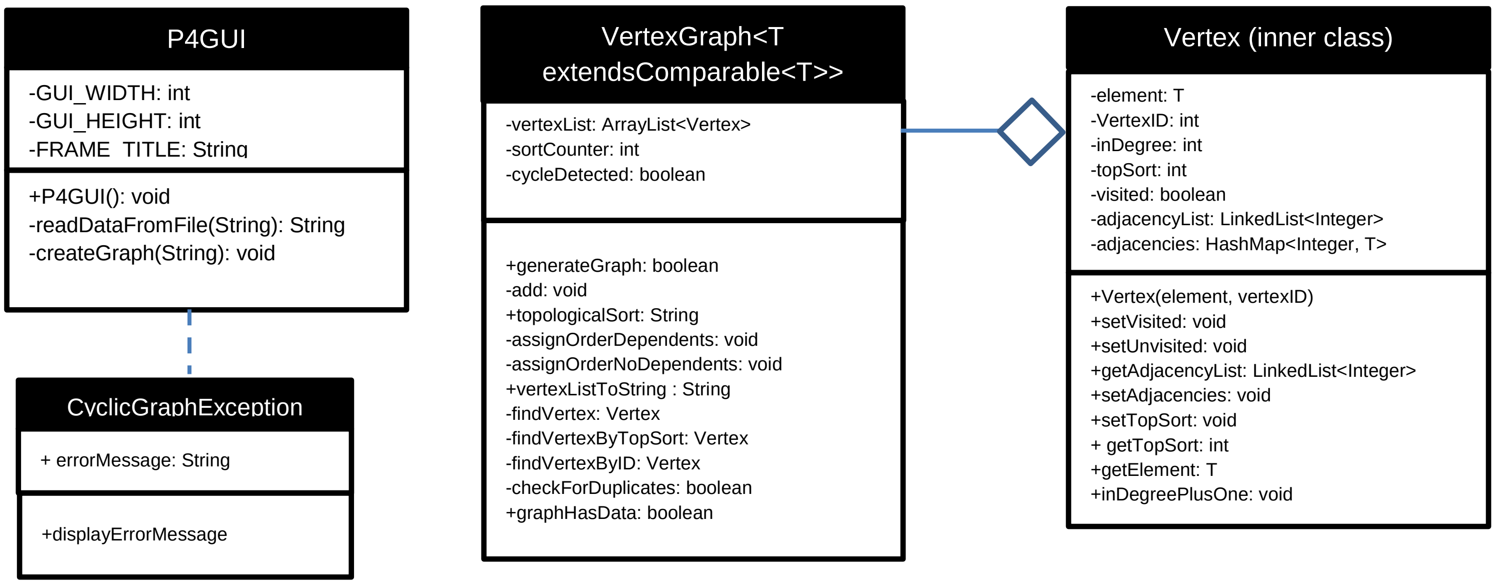
Eric Olsen

Assumptions:

The task for this assignment is to build a program that creates a graph data structure and performs a topological sort of its component vertices. The primary idea is that this program emulates the graph structure used by the Java command line compiler. We need to make several assumptions for this assignment. The first is that our vertex nodes are defined by String values which are passed from the GUI and processed as generic type parameters in a separate class. The second assumption is that our vertices and their dependencies are determined by a text file which is read into the GUI, parsed, and passed to a separate class for processing. Every vertex with dependent nodes is identified by its own line in the text file and its dependent vertices follow it by a single space delimiter. Third, our program will not allow cycles to occur because it makes performing a topological sort much more complicated. Finally, within our VertexGraph class, we will utilize a LinkedList of integers and a HashMap of type parameter to an integer VertexID to keep track of our elements.

Main Design Decisions:

Before describing the rest of this project in detail, the UML diagram is below and demonstrates the 4 classes that constitute this program.



P4GUI is responsible for all data displayed to the user. Its two JButtons and associated JTextFields perform the two methods mandated in the project specification: “Build Directed Graph” invokes the createGraph method in the VertexGraph class and “Topological Order” displays the order of class recompilation in the textPane as specified by the user in the “Class to recompile” textfield. P4GUI also uses a private readDataFromFile method to extract the contents of the attached text file. The createGraph method also takes the String from of the text file and creates an ArrayList of LinkedList String elements before passing this ArrayList to VertexGraph. Passing an ArrayList of LinkedList elements as opposed to a single String made generating the graph as a generic data type more tenable.

VertexGraph and its inner class, Vertex, merit some explanation. I will begin with Vertex. The inner class contains the required fields of its type parameter, integer vertexID, integer LinkedList of adjacencies, and a hash map of its vertexID to element mapping. I also added the fields inDegree for a default topological sort starting point, topSort for traversing the correct topological order, and visited for assisting when the graph is sorted later in the program. This inner class probably has a few too many inner setter and getter methods. Most of the internal methods were necessary because the outer class, VertexGraph, had to manipulate these fields in order to correctly add verticies, connect them, and sort them.

VertexGraph extends the Comparable Class primarily so we can invoke the compareTo method for comparing our element type parameters. This class also has three fields, an integer sortCounter, a boolean cycleDetected and an overall ArrayList for holding the values of all our instances of inner class Vertices. Originally I tried to implement all the vertex nodes as a HashSet so I could avoid checking for duplicate node values. In the end I still needed to conduct checks for pre-existing vertices because of their unique integer ID values which java would overwrite when adding duplicates. Using an ArrayList as the overall data structure holder has some disadvantages because a linear traversal of the data structure is required for every inserted vertex. Consequently, this data structure would not be suitable for situations where thousands of items were mapped together.

The generateGraph method receives the ArrayList of LinkedList String elements from the GUI. Here we process these Strings as type parameters. We first clear the existing graph and then process this ArrayList one line at a time. The add method handles each LinkedList per index of our ArrayList. As the add method receives each LinkedList, it takes the type parameter element and first checks to see if this element already exists. It does this with a private checkForDuplicates method that traverses the overall vertex ArrayList using the compareTo method. If no match is located, this method generates an integer vertexID which is the current size of the overall ArrayList. It also instantiates a Vertex object and adds this vertex to the overall ArrayList. During this first pass we do not yet worry about constructing our edges, we only care about creating all the nodes of the graph.

After adding all new vertex elements we begin a second iteration of the LinkedList. We first declare the vertex at index zero a rootVertex; every vertex following this element will have an edge that is dependent on this root. We generate a temporary LinkedList and add all dependent vertexID integer values to this list. When our traversal is complete, we add this list to the rootVertex's adjacencyList and conclude the add method until the next LinkedList arrives for processing.

TopologicalSort is a method that uses two helper methods to conduct a sort of the graph dependencies given an existing element within the graph. This method starts by setting the sortCounter integer to zero. It also sets all Vertex internal boolean values to false and their internal topSort values to -1. We then find the vertex associated with the parameter element and assign it as the rootVertex for this method. We then invoke the assignOrderDependents and recurseively visit every vertex that has dependent nodes. As we visit these nodes we increment the sortCounter, assign this counter as the vertex topSort value, and mark the vertex as visited. We then do this same operation to all vertices with no dependent vertices using the assignOrderNoDependents. After concluding both these methods, the topologicalSort method traverses the vertex ArrayList and generates a string for all verticies in ascending order that have a positive topSort value The method then returns this concantenated String to P4GUI.

Error Handling:

Avoiding a file read error in the GUI was fairly straightforward; I only needed to add a try-catch clause to the action event that read the text file. Detecting a cycle in the directed graph, however, took a concerted effort that spanned several methods within the VertexGraph class. We first needed to check for a vertex which listed itself as a dependent. We check for this during our adjacencyList generation in the add method. Next we needed to look at our topological sort method in order to catch cycles that listed a root node as dependent that was at least one degree separated. To detect this we utilize the inner class boolean values to see if the sorting algorithm had already visited the node being sorted. If we tried to sort a vertex which was already visited, the sorting algorithm throws our CyclicGraphException and terminates the sort.

On a final note, I thought the best way to detect cycles before taking user input was to actually conduct the topological sort starting with the first element in the first index of the overall ArrayList. The successful default sort and an internal check that the graph contains data is what returns a boolean value back to P4GUI. This boolean value determines whether the user sees a message success or failure message.

Program Testing:

Test Case #	Input	Description	Expected Output	Actual Output	Pass / Fail
1	ClassA ClassC ClassE ClassB ClassD ClassG ClassE ClassB ClassF ClassH ClassI ClassC	Best case (specification example).	Successful graph generation. Correct topological sort	Graph created. Sort from ClassA: ClassA ClassE ClassB ClassC ClassD ClassG ClassF ClassH Sort From ClassE: ClassE ClassB ClassD ClassG ClassF ClassH Sort from Class C: ClassC	Pass

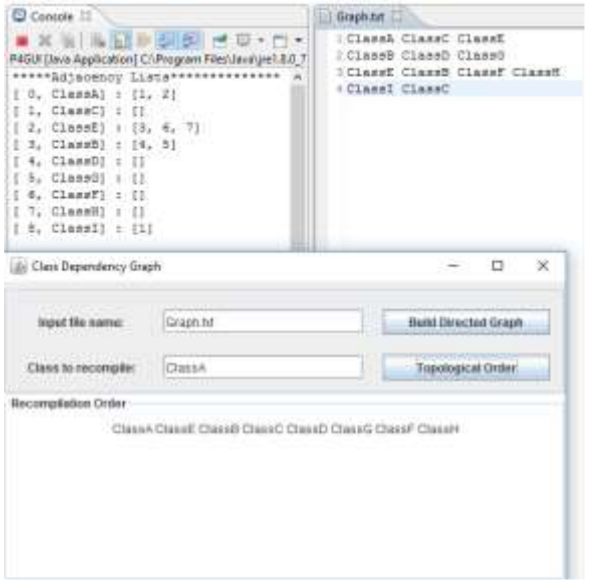
2	`CMSC330 CMSC335 CMSC495`
	`CMIS242 CMSC150 CMSC350`
	`CMIS141 CMIS242`
	`MATH141 CMSC150`
	`CMSC350 CMSC330`
		Alternate best case. In this case if UMUC rewrote a course, it would need to evaluate other classes requiring it as a pre-requisite	Successful graph generation. Correct topological sort.	Graph created.
				Sort from CMSC350: CMSC350 CMSC330 CMSC335 CMSC495
				Sort from MATH141: MATH141 CMSC150
				Sort from CMSC495: CMSC495
					Pass

3	ClassA ClassC ClassE ClassB ClassD ClassG ClassE ClassB ClassF ClassA ClassI ClassC	Check on graph cycle detection	Caught Exception	Exception Caught	Pass
4	CMSC350 CMSC150 CMSC350	Alternate cycle check. Typo in college course catalogue	Caught exception	Exception Caught	Pass
5	File name "grape.txt"	File read check	Caught exception	Exception caught	Pass

6	`ClassA ClassC`
	`ClassA ClassE`
	`ClassB ClassD`
	`ClassB ClassG`
	`ClassE ClassB`
	`ClassE ClassH`
	`ClassE ClassF`
	`ClassI ClassC`
		Additional dependent nodes; derived from test case 1.	Nearly identical output as test case 1	Graph created (near match to case 1)
				Sort from ClassA: ClassA ClassE ClassB ClassC ClassD ClassG ClassH ClassF
				Sort From ClassE: ClassE ClassB ClassD ClassG ClassH ClassF
				Sort from Class C: ClassC
					Pass

7	Type incorrect value for sort for best case "ClassZ"	Type an incorrect value for the sort value	Error message	Error message displayed	Pass

Test Case 1:



```
*****Adjacency Lists***************
[ 0, ClassA] : [1, 2]
[ 1, ClassC] : []
[ 2, ClassE] : [3, 6, 7]
[ 3, ClassB] : [4, 5]
[ 4, ClassD] : []
[ 5, ClassG] : []
[ 6, ClassF] : []
[ 7, ClassH] : []
[ 8, ClassI] : [1]
```

```
ClassA ClassC ClassE
ClassB ClassD ClassG
ClassE ClassB ClassF ClassH
ClassI ClassC
```

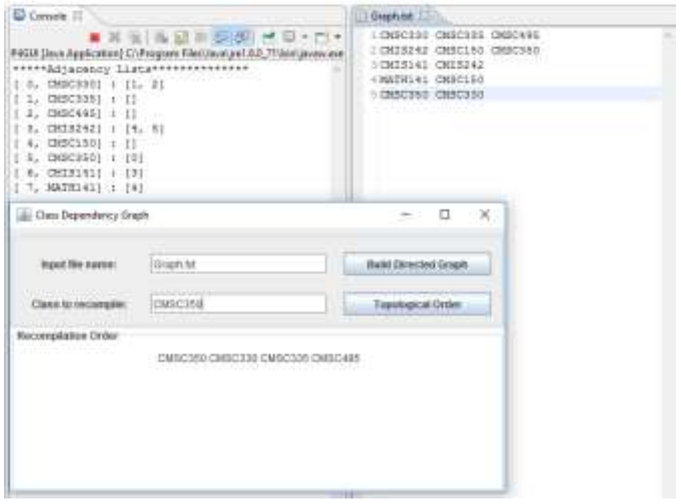
Input file name: Graph.txt

Class to recompile: ClassA

Recompilation Order

ClassA ClassE ClassB ClassC ClassD ClassG ClassF ClassH

Test Case2:



```
*****Adjacency Lists***************
[ 0, CMSC330] : [1, 2]
[ 1, CMSC335] : []
[ 2, CMSC495] : []
[ 3, CMIS242] : [4, 5]
[ 4, CMSC150] : []
[ 5, CMSC350] : [0]
[ 6, CMIS141] : [3]
[ 7, MATH141] : [4]
```

```
CMSC330 CMSC335 CMSC495
CMIS242 CMSC150 CMSC350
CMIS141 CMIS242
MATH141 CMSC150
CMSC350 CMSC330
```

Input file name: Graph.txt

Class to recompile: CMSC350

Recompilation Order

CMSC350 CMSC330 CMSC335 CMSC495

Lessons Learned:

This project taught me how complicated and useful linking vertex nodes together truly is. I felt very weak with the Java Generics class last week, but got a great deal of practice with it during this project. I think this overall data structure could be implemented much more efficiently as a set rather than a list, but conducting all the required error checks made this task very difficult.