John Kucera

Prof. Ioan Salomie

CMSC 350 Data Structures and Analysis

4 March 2020

Week 8: Project 4 Solution Description

**Class Dependency Graph GUI**

1. **Assumptions, Main design decisions, and error handling**

In my GUI, I have assumed that any characters and any other possible tokens can be entered. For the file name, no file will open and the program will not continue unless an existing text file is input in the “file.txt” format. Trailing white space is allowed when inputting the file name, but it is not allowed when entering the class to recompile. Any white space around the input class will cause an Invalid Class Name Exception. In addition, any input that does not match a class name from the respective file will raise the same Invalid Class Name Exception.

Assuming anything can be entered, I have tried to catch each possible error that could be entered through the GUI. This includes:

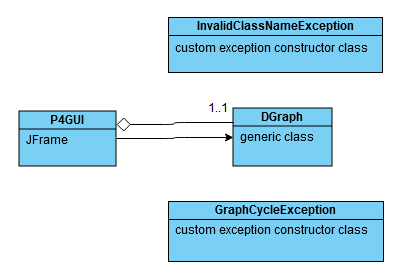
1. NullPointerException – for cases where the input is empty
2. InvalidClassNameException – for cases where there is any input that does match a class name from the respective open file, or there is white space
3. GraphCycleException – for cases where the created directed graph contains a cycle
4. IOException – for cases when the file fails to be read or the file name is invalid

As for the main design decisions, I decided to use the minimal implementation skeleton provided in Project 4 Indications for my DGraph generic class. In addition to the suggested components (DGraph default constructor, addVertex, addEdge, topOrdGeneration, buildDGraphFromFile), I added some methods of my own: tokenizeFile, getClass, and depthFirstSearch. The tokenizeFile method directly reads the input text file and tokenizes the class names while adding them to an ArrayList as strings. The getClass method uses the hashmap to return a string of a class name if given a vertex index. The depthFirstSearch performs a depth-first search algorithm on the graph given a starting vertex. Also, contrary to the suggestion in the Project 4 Indications, I created buildDGraphFromFile in the DGraph class rather than my P4GUI class. This is just because I prefer to keep all my data structure creation in their own class – I understand that the recommendation was to avoid casting operations, but I had no trouble concerning that with my program.

The way my program flows is like this: the user inputs a file name into P4GUI, then presses “Build Directed Graph”. First, tokenizeFile in DGraph.java reads the file and converts the class names into Strings while putting them into an ArrayList. DGraph.java contains private variables adjacencyList (an ArrayList of LinkedList of type Integer), neighborsList (List of Integers), mapTtoInteger (HashMap), recompOrder (StringBuilder), and numVertices (Integer). Next, buildDGraphFromFile creates a Directed Graph by using the addVertex method for each class (which are now tokens) and the addEdge method which connects two vertices. addVertex adds the token (class name) to the mapTtoInteger while creating an empty LinkedList of Integers called edgeHolder. edgeHolder is added to adjacencyList in the appropriate index for the current vertex. addEdge will add integers to the adjacencyList based on which classes (vertices) are dependent the current class (vertex), creating “edges” between those classes and the current class.

Next, the user inputs a class name into P4GUI, then presses “Topological Order”. The method topOrdGeneration in DGraph.java uses the input class as the starting vertex, throwing InvalidClassNameException and GraphCycleException to be caught in P4GUI. Then, depthFirstSearch is performed using the index of the input class (starting vertex), where the order of recompilation is appended to the recompOrder StringBuilder in topological order. Here, neighborsList is used to detect if a current vertex is already a neighbor, which means there is a cycle in the graph. When depthFirstSearch finishes recursing, the recompOrder StringBuilder is returned to topOrdGeneration. topOrdGeneration converts it into a normal String, and String recompOrder is then returned to P4GUI where it is displayed under “Recompilation Order”.

1. **UML Class diagram**



1. **Test Cases**

FILES I CREATED AND USED

file 1: graph.txt

ClassA ClassC ClassE

ClassB ClassD ClassG

ClassE ClassB ClassF ClassH

ClassI ClassC

file 2: graph2.txt

ClassZ ClassX ClassW ClassV

ClassV ClassT

ClassU ClassW ClassX

ClassX ClassR ClassQ ClassS

ClassQ ClassY

file3: graphWithCycle.txt

ClassJ ClassK

ClassM ClassJ ClassN

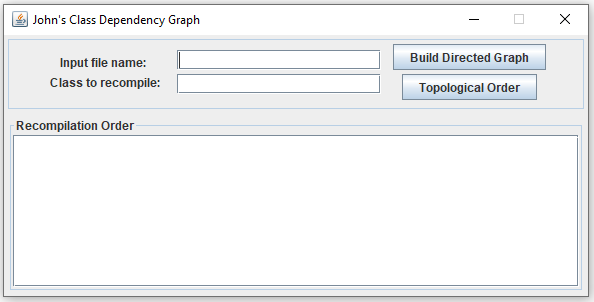
ClassN ClassO ClassM

ClassP ClassO

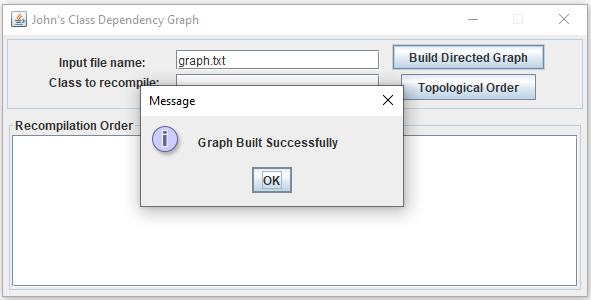
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **What aspect is tested** | **Input** | **Expected Output** | **Actual Output** | **Pass/**  **Fail** |
| If the user can close the GUI | Click “X” button on frame | GUI is closed and exited | GUI is closed and exited | PASS |
| (file 1) If a Directed Graph can be built from file | Input “graph.txt” into “Input file name” text field, press “Build Directed Graph” | Message Dialog Box: “Graph Built Successfully” | Message Dialog Box: “Graph Built Successfully” | PASS |
| (file 1) If correct topological order of graph can be generated, case 1 | Input “ClassA” into “Class to recompile” text field, press “Topological Order” | Under “Recompilation Order”: ClassA ClassC ClassE ClassB ClassD ClassG ClassF ClassH | Under “Recompilation Order”: ClassA ClassC ClassE ClassB ClassD ClassG ClassF ClassH | PASS |
| (file 1) If correct topological order of graph can be generated, case 2 | Input “ClassI” into “Class to recompile” text field, press “Topological Order” | Under “Recompilation Order”: ClassI ClassC | Under “Recompilation Order”: ClassI ClassC | PASS |
| (file 2) If a Directed Graph can be built from file | Input “graph2.txt” into “Input file name” text field, press “Build Directed Graph” | Message Dialog Box: “Graph Built Successfully” | Message Dialog Box: “Graph Built Successfully” | PASS |
| (file 2) If correct topological order of graph can be generated | Input “ClassZ” into “Class to recompile” text field, press “Topological Order” | Under “Recompilation Order”: ClassZ ClassX ClassR ClassQ ClassY ClassS ClassW ClassV ClassT | Under “Recompilation Order”: ClassZ ClassX ClassR ClassQ ClassY ClassS ClassW ClassV ClassT | PASS |
| (file 2) If correct topological order of graph can be generated | Input “ClassU” into “Class to recompile” text field, press “Topological Order” | Under “Recompilation Order”: ClassU ClassW ClassX ClassR ClassQ ClassY ClassS | Under “Recompilation Order”: ClassU ClassW ClassX ClassR ClassQ ClassY ClassS | PASS |
| Input invalid file name | Input “graphboy.txt” into “Input file name” text field, press “Build Directed Graph” | Message Dialog Box: “File Did Not  Open” | Message Dialog Box: “File Did Not  Open” | PASS |
| Input invalid file name (2) | Input “qwerty” into “Input file name” text field, press “Build Directed Graph” | Message Dialog Box: “File Did Not  Open” | Message Dialog Box: “File Did Not  Open” | PASS |
| Input invalid class name | (Directed Graph for graph.txt has been built) Input “ClassyA” into “Class to recompile” text field, press “Topological Order” | Message Dialog Box: “Invalid Class Name: ClassyA. Please try again.” | Message Dialog Box: “Invalid Class Name: ClassyA. Please try again.” | PASS |
| Input invalid class name (2) | (Directed Graph for graph.txt has been built) Input “cmsc350” into “Class to recompile” text field, press “Topological Order” | Message Dialog Box: “Invalid Class Name: cmsc350. Please try again.” | Message Dialog Box: “Invalid Class Name: cmsc350. Please try again.” | PASS |
| (file 3) Generate topological order of graph with cycle(s) | (Directed Graph for graphWithCycle.txt has been built)  Input “ClassM” into “Class to recompile” text field, press “Topological Order” | Message Dialog Box: “This Directed Graph contains a cycle due to circular class dependency.” | Message Dialog Box: “This Directed Graph contains a cycle due to circular class dependency.” | PASS |
| empty input (file name) | no input, press “Build Directed Graph” | Message Dialog Box: “Please enter a file name.” | Message Dialog Box: “Please enter a file name.” | PASS |
| empty input (class name) | (Directed Graph for graph.txt has been built)  no input, press “Topological Sort” | Message Dialog Box: “Please enter a class name.” | Message Dialog Box: “Please enter a class name.” | PASS |

1. **Screenshots of program execution**

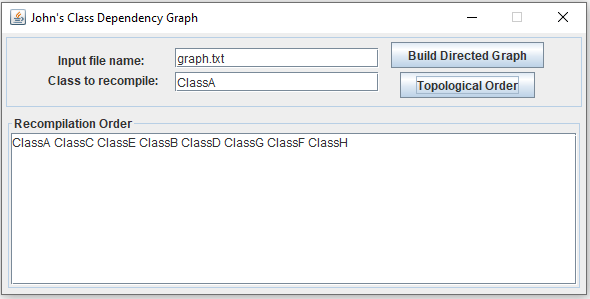
GUI:



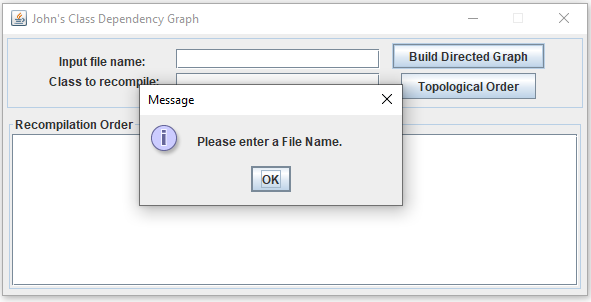
GUI, input correct file name and press Build Directed Graph button:



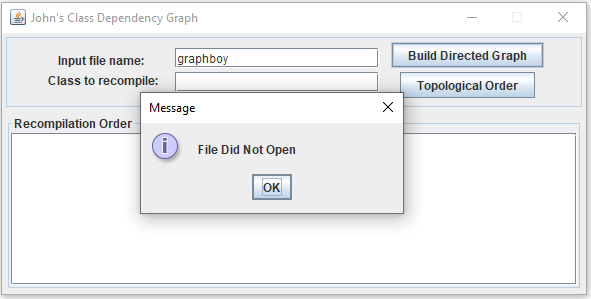
GUI, input correct class name and press Topological Order button:



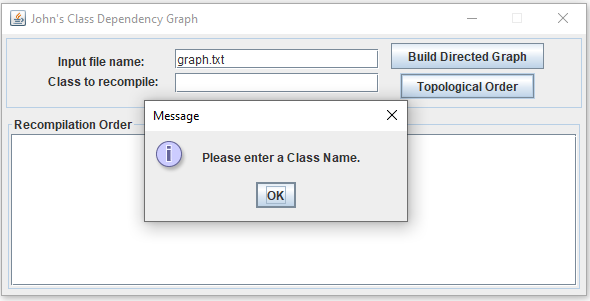
GUI, input nothing and press Build Directed Graph:



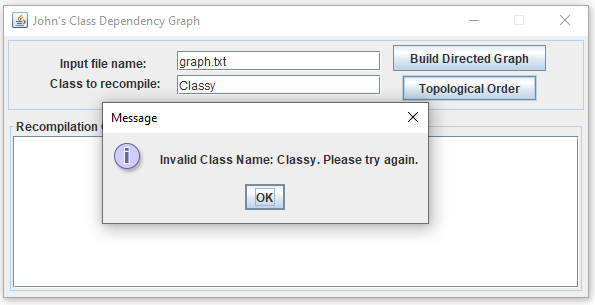
GUI, input INCORRECT file name and press Build Directed Graph:



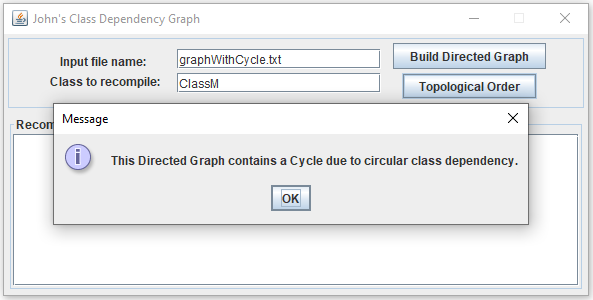
GUI, input nothing and press Topological Order (after Directed Graph is built):



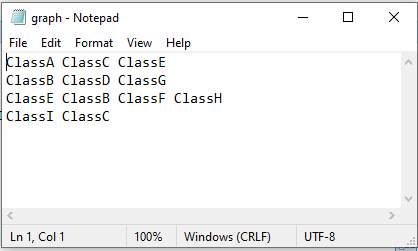
GUI, input INCORRECT class name and press Topological Order:



GUI, build Directed Graph and attempt to recompile a class that will cause a cycle:



Example of text file (graph.txt):



1. **Lessons learned**

This was my first time working with graphs in Java and thankfully, I found it to be a straightforward experience. This is mainly because of the experience I have had with other data structures in this class. The biggest lesson I have learned from this class overall, which I applied to this program, is that one of the most essential parts of data structures in computer science is giving connections to the elements. I have made many mistakes in previous projects where I created the elements but failed to connect them, which really means I didn’t actually create a data structure. A data structure MUST have connection between the elements to work, whether it is child/parent relationships in trees or edges in a graph. With this in mind, I really paid attention to the way addEdge method works in this program to ensure that the vertices are properly connected.

One lesson I learned this time around, which was my first time making this mistake, was trying to include the recursive depth-first search in the topOrdGeneration method. When I tried doing that, it just did not work out because I did not have a separate place to start an initialization of the depth-first search algorithm. It took me some rereading through my program and comparing it to previous recursive methods I have written, then finally realized that I needed to make the depth-first search its own method. Then, I called that method ONCE in topOrdGeneration method where it returned the StringBuilder for the recompilation order. I will always remember to do something like this in the future when writing other recursive orders.

While this is not a big lesson I learned, it was refreshing to go back to customizing my GUIs in more depth. The majority of projects in this class did not require GUIs with many components or complex positioning – just some text fields with labels and buttons next to them. For this GUI, however, I really had to use GridBagConstraints to properly position everything in the GUI to be like the example and even used a JScrollPane for displaying the Recompilation Order inside the bottom border. I know that this is beginner material, but I still very much enjoy having to customize my GUIs with different constraints, borders, and other unique components.