Project #3 Solution Description

Nicholas Mills

3/3/18

UMUC

CMSC 350

Ioan Salomie

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

Assumptions

* The file is in the root folder of the project.
* The file will follow the proper class naming structure.
* There are no more than 26 classes in a given file.
* There should be no text in the file other than the class names.
* Each line represents a group of classes which are dependent upon the first class on the line.
* The input will be no more than 30 characters. The program should still calculate properly, but the field will be too small to show the full values.
* The compile order of the classes matters only insofar as they properly build as their parents change.
* addVertex will only add a node if a node by that name does not already exist. There are no duplicate nodes.
* addEdge will only add an edge if the two nodes already exist. It will not create nodes on its own.

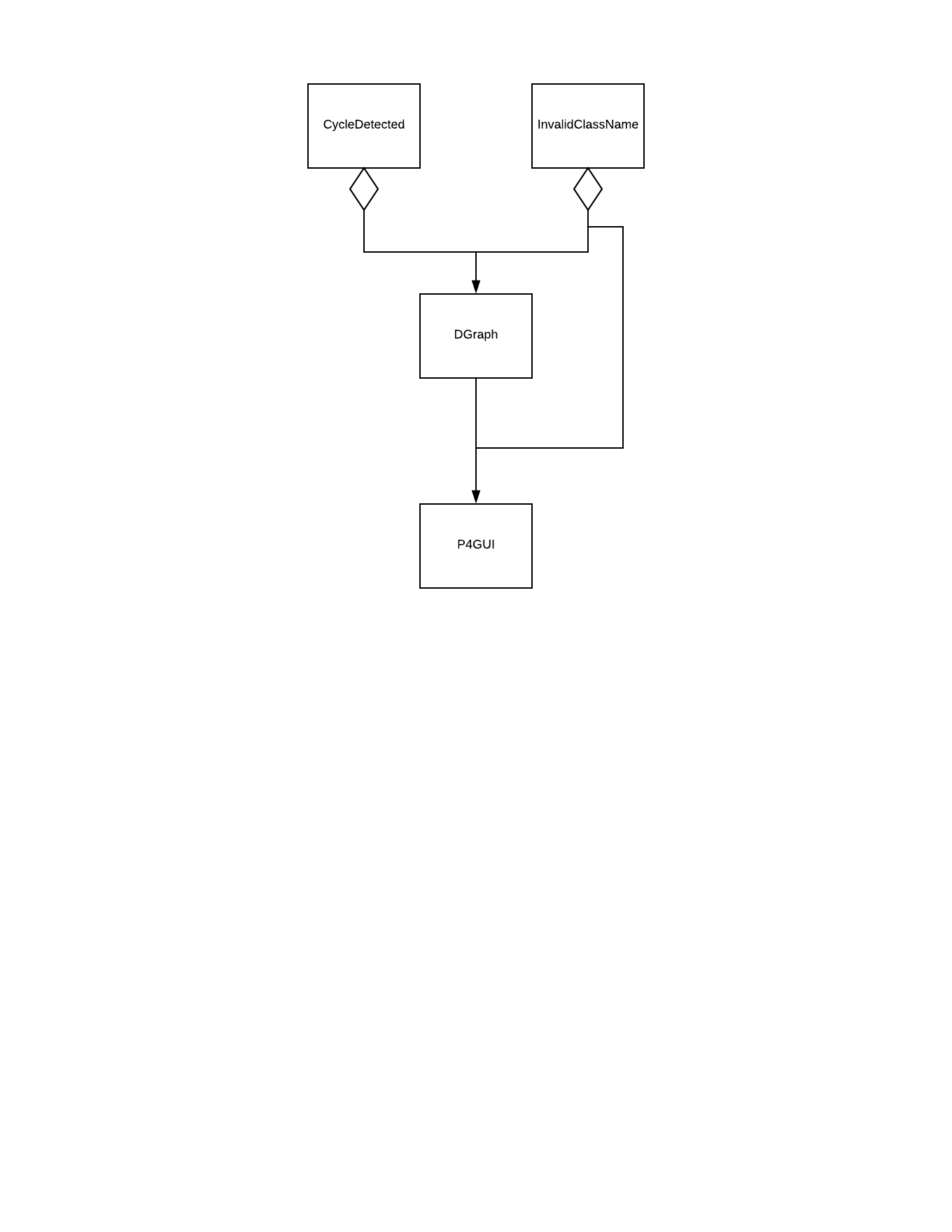
Main design decisions

* For tokenizing the file, I used two Scanners, one for the file and another one for each line. This allowed me to detect where a new line began. I scanned the file within the GUI class and then passed relevant information to the buildDGraph. I did this because DGraph is a generic class and I was uncertain how one was meant to create objects of type T through the tokens found in the file.
* For the GUI, I used a BoxLayout combined with 2 different panels to organize the different elements. I used a combination of a GridLayout for the inputPanel and a TitledBorder created in a BorderFactory for the outputPanel.
* For the topological sort, I used a recursive method, topOrdRecursive, with a helper method, topOrdGeneration, to delve into each relevant element of the adjacency list. This checked each of them for dependencies and added them to the list of classes that needed to be recompiled.
* I created a separate method, keyFromValue, to retrieve a key from the hashmap based on the value of said hashmap. Since the numbering mechanism is one-to-one, this retrieves the proper class name for output.
* addVertex adds a new node for the passed class if it does not already exist.
* addEdge adds a new edge between two nodes if they exist.

Error handling

* If the file contains class names that don’t follow the proper naming convention, an InvalidClassName exception is thrown and the user is informed.
* If the file cannot be found, a FileNotFoundException is thrown and the user is informed.
* If the specified class for the topological sort cannot be found, an InvalidClassName exception is thrown and the user is informed.
* If a class is found more than once in a list of dependencies, a cycle is detected and a CycleDetected exception is thrown. The user is informed.
* If an edge is attempted to be made and one of the classes does not exist, an InvalidClassName exception is thrown.

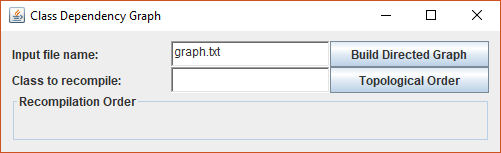
**2. UML class diagrams**

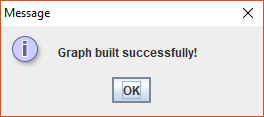


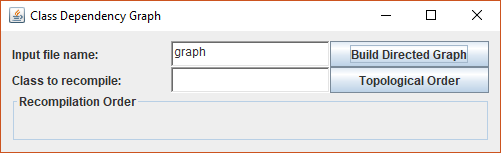
**3. Test cases**

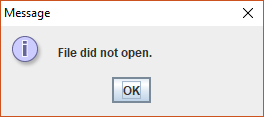
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **What aspect is tested** | **Input** | **Expected Output** | **Actual Output** | **Pass / Fail** |
| Build Success Dialog | graph.txt | Graph built successfully! | Graph built successfully! | **P** |
| File Read Error | graph | File did not open. | File did not open. | **P** |
| Standard Example | graph.txt  ClassA | ClassA ClassC ClassE ClassB ClassD ClassG ClassF ClassH | ClassA ClassC ClassE ClassB ClassD ClassG ClassF ClassH | **P** |
| Invalid Class Name | graph.txt  ClassT | Class not found. | Class not found. | **P** |
| Invalid Name Format | graph.txt(edited) | File contained an invalid class. | File contained an invalid class. | **P** |
| Cycle Detection | graph.txt(edited)  ClassA | Cycle Detected. | Cycle Detected. | **P** |

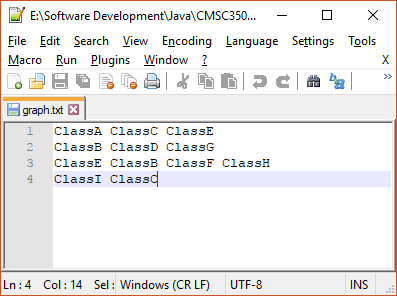
**4. Screenshots**

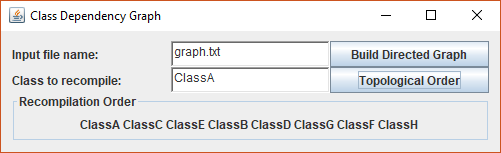


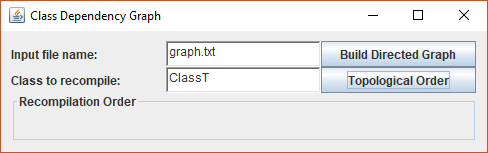


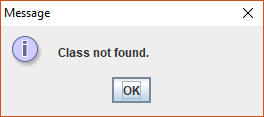


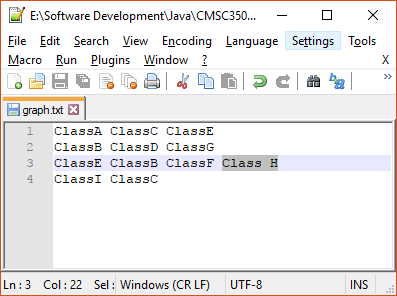


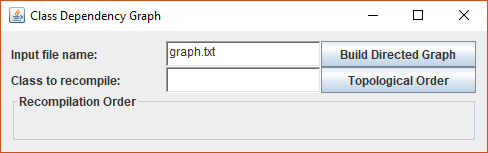


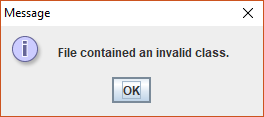


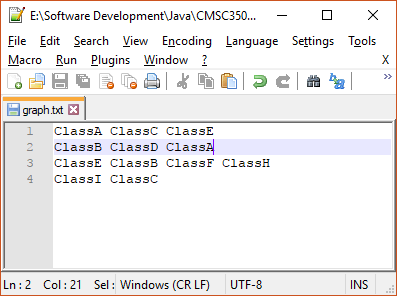


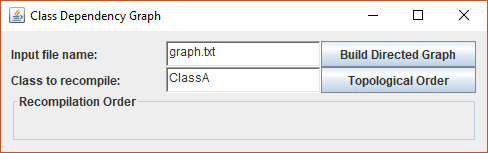


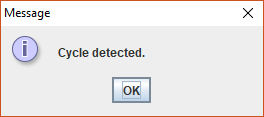












**5. Lessons learned**

This time around, I implemented some of the lessons that I had been taught in previous projects. I gave myself enough time to figure out the problem and I didn’t let myself get too hung up on the solution being exactly the way that I wanted it. I didn’t need the solution to be identical, just correct, which I believe that it is. I didn’t need to create a completely identical GUI, just one that’s close in appearance and functionality.

In addition to life lessons, I also was able to implement a number of lessons that I had learned previously in the class. I was able to implement recursive methods of my own design and more adequately, but still not perfectly, interact with generic classes. I created exceptions in the proper structure, though I’ve still yet to utilize them to the fullest extent and I can still use some work on my comment structure. I need to read up on Javadoc and improve that.

Most of all, I need to work on my generic class interaction. The major issues I was having in the last project were due to trying to deal with generics and one of the major issue of this one was also generic related. I also have a tendency to use inefficient, verbose, overly complex solutions. These are weak points in my coding performance and could use improvement.