CSCI 3901 Assignment 3

Due date: Friday, March 4, 2022 at 4:00pm Halifax time. Submissions through your CS gitlab repository in https://git.cs.dal.ca/courses/2022-winter/csci-3901/assignment3/????.git where ???? is your CS id.

Problem 1

Goal
Work with graphs.

Background
Large software projects can have thousands of files to contribute towards the project. When you change one file, all the files that depend on that file can be impacted by the change. When we compile that changed file, we must also recompile all the files that depend on it (and, often, then the files that depend on those other ones) to ensure that all our software is consistent.

One solution is to just recompile every file in the project when one file is changed. That solution wastes a lot of time in recompiling files that are not affected. The current solution is to first identify only those files that _could_ be affected and just recompile those.

Build environment tools map out these dependencies and identify the set of files affected when we change one file. These tools are built into your IDE for smaller project and into independent management programs for larger projects (make, ant, rake, and gradle are examples of these separate tools).

You will develop a Java class that will manage and report on the file dependencies in a project.

Problem
Implement a class called DependencyManager that will accept information about how different files depend on one another and will then report on the status of those files.

Your DependencyManager class will include the following methods, at a minimum:

- Constructor that accepts no arguments
- boolean addClass(String className) throws IllegalArgumentException
- boolean addClass(String className, Set<String> dependencies) throws IllegalArgumentException
- List<String> buildOrder () throws IllegalStateException
- Set<Set<String>> standaloneModules()
- List<String> highUseClasses(int listMax) throws IllegalArgumentException
- boolean hasDependencyCycle ()

The constructor sets up whatever your class needs. Each of the other methods operates as follows:

addClass

Define a class for the dependency manager to track.

- className – the class to add to the manager
- dependencies – if present, a set of modules that this current class depends upon

The boolean return value represents whether or not all class names provided are being tracked by the dependency manager after the method call ends.

If we ask to add a class with dependencies and then call addClass on that class with another set of dependencies then these second set of dependencies are added to any currently-tracked dependencies for the class.

The method throws an IllegalArgumentException if className cannot be represent a class or if the set of dependencies is unusable.

buildOrder

Return a sequence of all class names in an order for which we could compile the objects and, in that compilation, when we reach a class in the sequence then all of the classes that it depends upon have already been compiled. In other words, if class A appears in position X of the sequence then all the classes that A depends upon appear in a position less than X in the sequence.

For example, suppose that class A depends on classes C, D, and E; that class B depends on classes D, E, F, and G; and that both classes C and D depend on class H then one possible returned list would be

H, C, D, E, F, G, A, B

and another list would be

E, F, G, H, D, B, C, A

Such a list can only be provided if there is no dependency cycle in the set of dependencies. If there is a cycle of dependencies then throw the IllegalStateException.

Note that the output sequence is not always unique. There can be many possible sequences. You are to report one viable sequence.

standaloneModules

Identify each (minimal) group of class files that can be compiled into standalone executable programs.

A set of class files can be compiled into a standalone executable program if no class in the set depends on a class that is not in the set.

A group is minimal if there is no set of classes from the group (other than the empty set or the complete set) whose removal would still leave something that is a standalone executable program.

For example, suppose that class A depends on classes B and C; that class C depends on class D; and that class E depends on classes F and G.

- the set A, B, C is not a standalone executable program since C depends on D and D isn't in the group.
- The set A, B, C, D, E, F, G is a standalone executable program, but it is not minimal since removing E, F, and G still leaves A, B, C, and D that forms a standalone executable program on its own.
- The set A, B, C, and D is a minimal standalone executable program. The set E, F, and G is also a minimal standalone executable program.

The method returns a set of these minimal standalone executable programs. For the given example, the method would return

{ {A, B, C, D}, {E, F, G} }

It is a set of sets. The set of sets is unique. The order of sets or of classes within a set doesn't matter (as understood by us using the Set ADT).

highUseClasses

Report the classes that the most other classes directly depend upon. Report the top "listMax" classes, in decreasing order of dependencies and, when there is a tie in those counts, in alphabetic order of the class name.

For example, suppose that we have a system with the following classes

A with 3 other classes depending directly on A
B with 8 other classes depending directly on B
C with 10 other classes depending directly on C
D with 8 other classes depending directly on D
E with 2 other classes depending directly on E
F with no other classes depending directly on F

Then highUseClasses (1) returns the list [C] since C is most used by others.
highUseClasses(4) returns the list [C, B, D, A]

If the listMax limit would stop the returned list part-way through a list of classes that all have the same number of classes depending on them then we return more than listMax classes to finish that list of classes with the same dependencies.

For the previous example, highUseClasses(2) would return the list [C, B, D]; since classes B and D have the came counts, we complete the returned list with both of them.

The method throws an IllegalArgumentException if listMax doesn't make sense for the request.

hasDependencyCycle

A dependency cycle happens when one class indirectly depends on itself. For example, if class A depends on class B, class B depends on class C, and class C depends on class A then class A indirectly depends on itself.

Return true if there exists one or more dependency cycles and return false if there are no dependency cycles.

Algorithms

Some of the methods build on standard graph algorithms that you can reference online. You do not need to invent algorithm ideas; you are allowed to use the ideas of algorithms that you find (but not the code). Don't forget to reference any algorithms that you use. Also, using the ideas doesn't mean that you're looking at an implementation of the algorithm and re-writing it in your style.

buildOrder	The nearest problem you will find is called a topological sort algorithm. However, you can also solve the problem using a variant of the post-order traversal ideas that we have seen in class.
standaloneModules	The problem breaks down to identifying each connected component of the graph. Again, traversal algorithms can help you with this problem.
hasDependencyCycle	There are a number of algorithms to detect cycles in graphs. Note that this method doesn't ask you to list them or to count them. As soon as you find a cycle, you have your answer. A typical approach is to do depth first traversals in your graph where you detect a cycle when your depth first search has you return to a vertex that you have already visited in your current depth first search path.

Assumptions

You may assume that

- All class names are lower case

Constraints

- You may use any data structures from the Java Collection Framework.
- You may not use a library package to for your graph or for the algorithm on your graph.
- If in doubt for testing, I will be running your program on timberlea.cs.dal.ca. Correct operation of your program shouldn't rely on any packages that aren't available on that system.

Notes

- Consider what kind of graph / graph characteristics underlie the problem. Choose your graph representation accordingly.
- You might consider writing your own method to print the graph. It might help simplify your debugging.
- Pick the order in which you will implement your methods. Implement the simpler ones first to get familiar with your graph representation. buildOrder will be the most challenging of the methods to write.

Marking scheme

- Documentation (internal and external), program organization, clarity, modularity, style – 4 marks
- Thorough (and non-overlapping) list of test cases for the problem – 3 marks
- Efficiency of your data structures and algorithms for the given task, as explained by your own description of that efficiency (part of external documentation) – 2 marks
- Marks for each of the requested methods:
  - addClass – 3 marks
  - buildOrder – 8 marks
  - standaloneModules – 8 marks
  - highUseClasses – 3 marks
  - hasDependencyCycle – 4 marks