Lab #4

***CSE2331/5331* (*Spring 2013*)**

Due Date: Sunday, April 21 by 11:59pm

Guidelines

- Any lab that does not compile will receive a **ZERO**. The grader will not do this for you when grading.
- All make-ups for lab must be accompanied by a documented and verifiable excuse well before the deadline. Given the severity of the emergency please inform me as soon as possible.
- Lab submissions will NOT be accepted via email to me or the grader.
- Work on the lab on your own, i.e. individually. **No group work**!
- ***All suspected cases of academic misconduct will be reported to the University Committee on Academic Misconduct for review.***

Objectives

Explore graph data structures and algorithms. Study the material we covered on Graphs and associated algorithms (Chapter 22 of the textbook and class notes). You can easily find quick tutorials regarding makefiles online if you need a quick refresher.

Materials

Implement your solution using either *Java* or *C++*. The grader will grade *Java* solutions using the *Eclipse IDE* for *Java* (I highly recommend this choice) installed on our *Windows* lab computers and grade *C++* solutions using the GNU *g++* compiler installed on our *stdlinux* system. *If you develop your code at home using a different IDE or compiler than what is available on our systems, I highly suggest you port your code to one of the platforms the grader will use and ensure your code compiles and runs properly before you submit***.** You can download the Eclipse IDE from www.eclipse.org for free.

Initial code is provided to you on Carmen (download the appropriate zip file). The grader will use the methods(functions) I have provided to grade your work. For those using *Java*, I have provided JUnit test code. *C++* coders can take a look at all the *Java* code I provide to get an idea on how to time and test your code in a similar fashion.

Coding

Your program will accept input in the following format:

```
Class1.java::Edit Class1.java
Class1.class:Class1.java:javac Class1.java
Class2.java::Edit Class2.java
Class2.class:Class2.java:javac Class2.java
MyApp.jar:Class1.class Class2.class:jar cvf *.class
```

Each line above has three parts – a target name, a list of targets that must be built before this target, and a command that is used to generate the target. Targets can have zero or more dependencies. In the example above, `Class1.java` has no dependencies, and `MyApp.jar` has two – it needs both `Class1.class` and `Class2.class` to exist before it can be built.

Your program will be responsible for parsing this input and creating a graph in memory. It will manipulate the graph to produce a list of commands that need to be executed to produce a given target. For example, the list of commands that must be executed to produce `Class1.class` is:

```
Edit Class1.java
javac Class1.java
```

This is the only acceptable list of commands in this case. Notice that there will not always be a unique correct answer. If we list the commands needed to produce `MyApp.jar`, there are several sequences that can work. Two of these would be:

```
Edit Class1.java
javac Class1.java
Edit Class2.java
javac Class2.java
jar cvf *.class
```

and

```
Edit Class1.java
Edit Class2.java
javac Class1.java
javac Class2.java
jar cvf *.class
```

Guidelines

Implement the ***Builder*** class provided to you (skeleton code on Carmen) by implementing each of the listed methods(functions). You will input a *makefile* (in the form of a *String* and not a stored file) to the constructor method *Builder*. After you create a Builder object you can then call the *makeTarget* method with a target name. The method will output a list of commands. The *makeTarget* method is responsible for providing just one correct output.

Your solution must conform to the following constraints (***Note, not adhering to all of these criteria exactly will result in a drastic reduction in your grade!***):

- Choose either the programming language *Java* or *C++* for your implementation.
- Insert your solution into the given code skeleton. Provide instance variables and constants (if any) and make them all private. Provide solutions for all methods listed in the class.
- Do not edit any class definition line I have provided in any way, e.g. do not change "public class Builder".
- Do not change any of the method (function) signatures in the classes. These are all public and non-static.
    - You may define additional *helper methods*, but these must be *private* inside the ***Builder*** Class.
- Make all instance variables *private*.
- You may define your own class(es).
- You must implement your ***OWN*** data structure to represent a graph. You cannot use a library or third party source that already implements one. Remember, ***ALL*** code must be your own.
- No global variables outside any class! This is especially true for those using C++. All variables must be either instance variables (private as mentioned above) or local to a method(function).
- The *Builder* constructor will throw a *CycleDetectedException* if the graph contains a cycle.
- The *Builder* constructor will throw an *UnknownTargetException* if a dependency does not have an associated target line.
- Other errors on the input will cause the *Builder* constructor to throw a *ParseException* (See my tests).
- The *makeTarget* method(function) will never return *null*.
- Do NOT add code to any of the exception classes.
- If you have any questions or concerns about how you will implement your solution then please ask me early!
- Document your code well. This means put your **name**, **date**, and **brief description of the class**, i.e. data structure, at the top. Provide multi-line comments for each method(function) and then single line comments in your code where appropriate (You do NOT need a comment for almost every line of code!). Provide single line comments for sections of code that represent a discernible algorithm. You will receive point reduction for too few or too many comments.
- Your solutions to all methods(functions) must be efficient in both storage and run-time.

Hints

1. One of the things you will need to do is perform a topological sort on the graph.
2. Make sure you consider what can go wrong with the *makefile* and with the processing of the graph, and that you throw appropriate exceptions.

Grading

- Source code (95%) – Design and implement your solution well using good Object Oriented design. Code passes all tests I have provided and of course any you dream up. The grader has another collection of tests that your code will be run against.
- Documentation and code readability (5%) – This includes good indentation and descriptive variable and function names.

Submission

Create a single zip file with the following work: 1) README file that includes how to compile and run your source code, 2) Well documented source code (Do NOT include executables or any compiled code).

Submit the zip file to the Carmen dropbox.