CSCI 2600 — Principles of Software

Homework 4: Designing and Implementing an ADT

Due: Thursday, Mar. 20, 2025, 11:59:59 pm

Read this document **entirely** before starting to work on your homework assignment. As always, you must commit and push to your Git repository, then submit on Submitty.

Submission Instructions

- Follow the directions in the version control handout for cloning your hw04 git repo. The URI that you will need to use to clone your personal repo for this homework would have the form of `https://submitty.cs.rpi.edu/git/s25/csci2600/hw04/RCSID` where RCSID is your RCS ID.
- Be sure to commit and push the files to Submitty. Follow the directions in the version control handout for adding and committing files.
- Be sure to include any additional files in your repo using **Team** → **Add to Index**.
- **Important:** You must press the **Grade My Repository** button, or your answers will not be graded.

Introduction

In this assignment, you will design, implement, and test a graph ADT. Given an abstract, high-level description of the desired ADT, you will develop a working Java implementation. You get to choose both the public interface and internal representation, then decide what unit tests to write to demonstrate that your implementation works correctly.

This assignment is the first part of a multi-part project. In subsequent assignments you will build several clients of your graph ADT.

Building a Graph

Definitions and Terminology:

A graph is a collection of nodes (also called vertices) and edges (also called links). Each edge connects two nodes. In a directed graph, edges are one-way: an edge e = ⟨A,B⟩ indicates B that is directly reachable from A. To indicate that B is directly reachable from A and A from B, a directed graph would have edges ⟨A,B⟩ and ⟨B,A⟩.

The children of node B are the nodes to which there is an edge from B. In Fig. 1, the children of b are a and c. Similarly, the parents of B are the nodes from which there is an edge to B. In Fig. 1, b only has one parent, a.

A path is a sequence of edges ⟨node1,node2⟩, ⟨node2,node3⟩, ⟨node3,node4⟩, ... In other words, a path is an ordered list of edges, where an edge to some node is immediately followed by an edge from that node. In Fig. 1, one possible path is ⟨b,a⟩, ⟨a,b⟩, ⟨b,c⟩. This path represents traveling from b to a to b to c. A path may traverse a given edge more than once.

A reflexive edge is an edge which starts and ends at the same node. In Fig 1. there is a reflexive edge from a to a: ⟨a,a⟩. Your graph implementation must allow for reflexive edges.

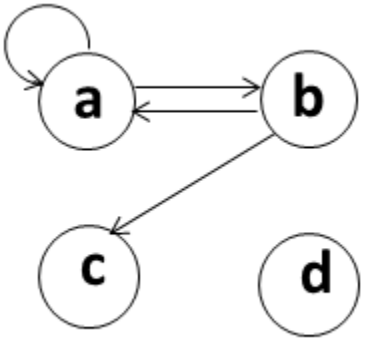

Figure 1: A simple directed graph with four nodes.

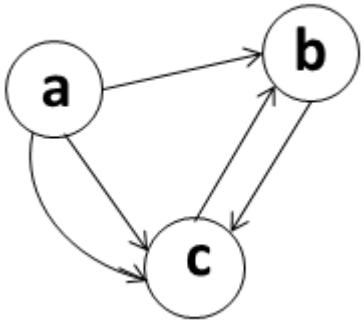

Figure 2: A directed multigraph.

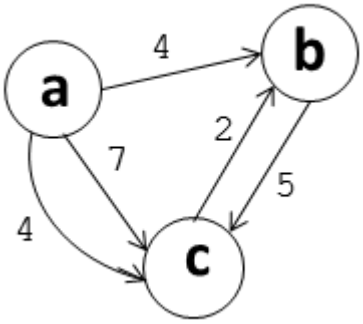

Figure 3: A directed labeled multigraph.

In a multigraph, there can be any number of edges (zero, one, or more) between a pair of nodes. Fig. 2 shows a multigraph with two edges from a to c.

In a labeled graph (Fig. 3), every edge has a label containing information of some sort. Labels are not unique: multiple edges may have the same label. In theory, a graph could contain two "identical" edges with the same starting point, ending point, and label, but for this project you may decide whether to allow identical edges in your implementation. (Whatever you decide, make sure it is clearly documented, so that clients understand how they can use your ADT.)

If you want to learn more, read Wikipedia's definition of a graph. Then if you still have a question, ask us.

Many interesting problems can be represented with graphs. For example:

- A graph can represent airline flights between cities, where each city is a node and an edge ⟨A,B⟩ indicates that there is a flight from A to B. The edge label might represent the cost in money (airfare), time (length of flight), or distance.
- To find walking routes across RPI's campus in Troy, you can build a graph where nodes represent buildings and other locations and edges represent walking paths connecting two locations. The label/cost of an edge is the physical length of that path.
- The Web can be modeled as a graph with node for every Webpage and an edge ⟨A,B⟩ if page A links to page B. The label could indicate the anchor text for a link on page A, or the number of links from page A to page B.
- Facebook is essentially a giant graph with nodes for users and edges between friends. (You can see a visualization of the Facebook graph.)

Problem 1: Write a Specification for Graph

To start, you will specify the API for a Java class or classes representing a directed labeled multigraph. The API, or public interface, is the collection of public classes and methods that clients of your graph ADT will use. We recommend that you work in iterative layers of detail. Start rough — preferably with pencil and paper — by brainstorming what operations the ADT needs to support, how it might be logically broken into modules (classes/interfaces), and how these modules connect to each other. Then, jot down a list of methods for each class that provide these operations. Think through some possible client applications, particularly finding efficient routes between different nodes, to get an idea about what operations might be needed.

Initially you should keep your design rough — don't write formal class and method specifications with all the proper clauses right away. Your design will likely go through multiple iterations, and it's easier to throw away parts before you've invested too much effort in them.

Once you are satisfied with your high-level design, write a specification for each class and method. Follow the convention we have been using in this class, remembering to use both standard Javadoc tags (`param, returns, throws`) and the ones introduced for this course (`requires, effects, modifies, etc.`) Go over the slides for specifications again. A good approach is to create skeleton implementations of your classes containing only method "stubs", then write your specifications in place in the code. A stub is a not-yet-implemented method whose body simply throws an exception, as you saw in the Homework 3 starter code. Stubbing out a class gives you the flexibility to write client code and tests that use it before completing the implementation. The skeleton code will not behave correctly or pass tests, but the program will compile and run.

For this assignment, you may restrict your graph to store the data in nodes and edge labels as `String`s. We strongly recommend you DO NOT define your classes as generic for this assignment. In a future assignment, you will use generics to make your ADT work with other data types — text, integers, doubles, etc. You may assume nodes are uniquely identified by their data contents: that is, no two nodes store the same data.

Design problems such as this one are open-ended by nature: we are not looking for one "right" design. There are principles of good design you have learned in lecture, however. You will also find suggestions of what to consider and how to approach the problem in the hints section. Also, designs naturally evolve as requirements change, so try to make your code extensible, but don't over-generalize.

Once you are done with the specification of your Graph ADT (overview, AF, and representation invariant and Principles of Software specifications for each abstract operation) copy and paste your specs and stubs into the file `answers/hw4_problem1.pdf`. You can use `RatNum.java` from Homework 3 as a guide.

Problem 2: Write Tests for Graph

Write a black-box test suite for your Graph specifications. It is important to write your tests before your code, but your tests will not pass until you have completed Problem 3.

- **Documentation**: Include with your test cases one short paragraph of documentation explaining your testing strategy, and which (if any) testing heuristics you used. Place this write-up in `answers/hw4_problem2.pdf`.

Create and commit new JUnit test classes in directory `src/test/java/hw4`. You must have a JUnit test case for `GraphWrapper` placed into file `src/test/java/hw4/GraphWrapperTest.java` which should already exist in your repository. (`GraphWrapper` is explained in Problem 3.) You may need to add other files later on. To create a new test case, right-click on `hw4` under `src/test/java`, then New → JUnit Test Case, then type a name for the new test class. Don't forget to add JUnit to the build path! Look back to Homework 0 and Homework 3 as examples of extensive test suites.

Problem 3: Implement Graph

A Java source code file `GraphWrapper.java` for the `GraphWrapper` class (it is explained shortly after) should already exist in your repository. When you later need to add other Java source code files, right-click on `hw4` under `src/main/java`, then New → Class, Package: `hw4`, and then enter the name of the class to be created. Be sure to use `Team → Add to Index` to add the file to your repo.

There are many ways to represent a graph. Here are a few:

- As a collection of edges.
- As an adjacency list, in which each node is associated with a list of its outgoing edges.
- As an adjacency matrix, which explicitly represents, for every pair ⟨A,B⟩ of edges, whether there is a link from A to B, and how many.

(1) In one brief paragraph, describe your representation. For at least the three representations above and your representation (if it is not one of the ones above), explain the advantage of that representation in a sentence. For example, do certain representations have advantages of efficiency in search or ease of implementation? Given these advantages, briefly explain why you chose the representation you did. You should place this discussion in a file called `answers/hw4_problem3.pdf`.

(2) Place a proper **abstraction function and representation invariant** for your Graph data type (and any other ADTs you create) in your source code. Also implement a private `checkRep()` method, which will help in finding errors in your implementation. You do not need a representation invariant or abstraction function for `GraphWrapper.java`. It is not an ADT.

(3) Provide an implementation of your graph data type. We ask that you strive first for a good design before worrying about performance right now. Eventually, however, your path-finding application will create and operate on very large graphs, so the scalability of your Graph implementation will be important. Since the path-finding algorithm must frequently look up the children for a given node, this operation should be performed quickly even for large graph sizes. Your graph building operations should also be reasonably efficient. As your implementation will likely use classes in the Java Collections Framework, you should understand the computational complexity of classes such as HashMap and ArrayList.

(4) Be sure to call your `checkRep()` where appropriate.

(5) Once you've finished your implementation, you should think about whether or not new tests are needed in addition to those you wrote before you started coding. If so, you should add these to your test suite. You should **append to the end of your test strategy writeup** in Problem 2 a description of any new tests you added, or why you feel that your original tests alone are sufficient. At this point you will use **EclEmma** (available through the Eclipse Marketplace), which measures your code coverage. Look for coverage of package `hw4` in EclEmma. You must achieve 75% or higher instruction (statement) coverage for all classes in package `hw4` except for `*Test` classes to receive full credit on this part. We measure the coverage of your Java code as executed by your test suite.

(6) If EclEmma is not already installed in Eclipse, select `Help` from the main menu and then select `Eclipse Marketplace`. In the search box, enter `EclEmma`. Select `Install` to begin the installation.

You have the freedom to design the Graph ADT as you wish (you can name your class, include operations at your discretion, and choose the rep as you like). However, we require that you implement an additional class, `GraphWrapper`, which has an instance field of the type of your Graph ADT, and implements the following public methods. You don't have to write the rep invariant, abstraction function, or `checkRep()` for `GraphWrapper`. `GraphWrapper` does not represent an ADT, it simply is an interface to your Graph ADT. You do need to write a test suite `GraphWrapperTest.java` in directory `src/test/java/hw4`. The purpose of `GraphWrapper` is to allow us to test your Graph ADT through a common interface. Below is the abstract description of operations in `GraphWrapper`.

`public GraphWrapper()`: This is the constructor of GraphWrapper. It initializes the instance field with a new empty instance of your Graph ADT.

`public void addNode(String nodeData)`: Adds a node represented by the string `nodeData` to your graph. If an identical node already exists in the graph, the output of `addNode()` is undefined, that is, it is left at your discretion.

`public void addEdge(String parentNode, String childNode, String edgeLabel)`: Creates an edge from `parentNode` to `childNode` with label `edgeLabel` in your graph. If either of the nodes does not exist in the graph, the output of this command is undefined. If an identical edge (same parent, child, and label) already exists, the output of this command is also undefined, as it is left at your discretion whether to allow identical edges in your implementation.

`public Iterator<String> listNodes()`: This operation has no effect on your graph. It returns an iterator which represents the nodes in lexicographical (alphabetical) order. As an example, for the graph shown in Fig. 3 above, a call to `listNodes()` should return an iterator that goes through the following strings:
`"a"`
`"b"`
`"c"`

`public Iterator<String> listChildren(String parentNode)`: This operation has no effect on your graph. It returns an iterator which represents the list of `childNode(edgeLabel)` in lexicographical (alphabetical) order by node name and secondarily by edge label. `childNode(edgeLabel)` means there is an edge with label `edgeLabel` from `parentNode` to `childNode`. If there are multiple edges from `parentNode` to some `childNode`, there should be a separate entry for each edge. If there is a reflexive edge, `parentNode(edgeLabel)` should be in the list. As an example, for the graph shown in Fig. 3 above, a call to `listChildren("a")` should return an iterator that goes through the following strings:
`"b(4)"`
`"c(4)"`
`"c(7)"`

`public String listChildrenXML(String parentNode)`: This operation has no effect on your graph. It returns a String which represents the list of child nodes and edges in XML format in lexicographical (alphabetical) order by node name and secondarily by edge label. `nodes` represents a top-level "root" container that includes all other elements placed between the opening `<nodes>` tag and the closing `</nodes>` tag. `<edge␣label="edgeLabel"␣node="childNode"␣/>` means there is an edge with label `edgeLabel` from `parentNode` to `childNode`. If there are multiple edges from `parentNode` to some `childNode`, there should be a separate entry for each edge. If there is a reflexive edge, `<edge␣label="edgeLabel"␣node="parentNode"␣/>` should be in the list. As an example, for the graph shown in Fig. 3 above, a call to `listChildrenXML("a")` should return the following multi-line string (double quotes surrounding the string are omitted for brevity) where indentation is represented by two spaces ␣␣:

```
<nodes>
  <edge label="4" node="b" />
  <edge label="4" node="c" />
  <edge label="7" node="c" />
</nodes>
```

Do not implement the graph ADT in GraphWrapper.java. Create a separate class for your graph.

Do not implement string formatting required by `listChildren()` and `listChildrenXML()` in your Graph ADT code. The specific formatting, like using parenthesis or XML markup should belong to `GraphWrapper`. Your Graph ADT design should be general enough to support all required operations but specific details of listing child nodes in a certain format should be the responsibility of `GraphWrapper`. In other words, your Graph ADT should provide operation(s) that both `listChildren()` and `listChildrenXML()` from `GraphWrapper` use without implying any specific formatting of the output.

Collaboration (0.5 pts.)

Please answer the following questions in a file named `hw4_collaboration.pdf` in your `answers/` directory.

The standard academic integrity policy applies to this homework.

State whether or not you collaborated with other students. If you did collaborate with other students, state their names and a brief description of how you collaborated.

Reflection (0.5 pts.)

Please answer the following questions in a file named `hw4_reflection.pdf` in your `answers/` directory. Answer briefly, but in enough detail to help you improve your own practice via introspection and to enable me to improve Principles of Software in the future.

- In retrospect, what could you have done better to reduce the time you spent solving this homework?
- What could we, the teaching staff, have done better to improve your learning experience in this homework?
- What do you know now that you did not know before beginning the homework?

  We will be awarding up to 1 extra credit point (at the discretion of the grader) for particularly insightful, constructive, and helpful reflection statements.

Grade Breakdown

- Compilation of code and test cases: 2 pts. (auto-graded)
- Quality of test suite, percent of your tests passed: 6 pts. (auto-graded)
- Quality of test suite, percent instruction (statement) coverage: 12 pts. (auto-graded)
- Instructor GraphWrapper tests: 10 pts. (auto-graded)
- Specifications (`answers/hw4_problem1.pdf`): 7 pts.
- Answers to Problem 2 questions (`answers/hw4_problem2.pdf`): 2 pts.
- Answers to Problem 3 questions (`answers/hw4_problem3.pdf` and your code): 10 pts.
- Collaboration and reflection: (0.5 pts. each)

Hints

Writing Specifications

To give you some sense of the kinds of issues you should be considering in your design, here are some questions you might want to consider. These don't, in general, have simple answers. You'll need to exercise judgment and think carefully about how decisions you make interfere with each other.

- Will the graph be mutable or immutable?
- Will the graph be implemented as a single class, or will there be a Java interface for the Graph specification and a separate class for the implementation?
- Will edges be objects in their own right? Will they be visible to a client of the abstract type?
- Will nodes be objects in their own right? Will they be visible to a client of the abstract type?
- When will the user specify the nodes and/or edges in the graph? (In the constructor? With an insertion method? Both? Can the user add multiple nodes and/or edges at once?)
- What kind of iterators will the type provide?
- Will the type provide any views, like the set view returned by the `entrySet()` method of java.util.Map?
- Will the type implement any standard Java collection interfaces?
- Will the type use any standard Java collections in its implementation?

In choosing what operations/methods to include, strive to include enough that the ADT will be convenient and useful for a client, but avoid the temptation to write an "everything but the kitchen sink" API. It is better to design a minimal than a maximal API. In the

real world, you can always add methods later. However, you can never remove them from a published API, and such methods may over-constrain the implementation in the future.

Make good use of the course staff. If you have concrete questions, then take your specification to office hours to get some feedback on your design and style. This is likely to save you a lot of time!

Working Incrementally

Although it is generally a bad idea to start coding before you have thought deeply, it often makes sense to work incrementally, interleaving design and coding. Once you have a sketch of your specification, you may want to write some experimental code. This should give you some concrete feedback on how easy it is to implement the methods you've specified. You may even want to start at the end, and write the code that uses your type, so that you can be confident that the methods you provide are sufficient.

This strategy can backfire and degenerate into mindless hacking, leaving you with a pile of low-quality code and an incoherent specification. To avoid that, bear three things in mind:

(1) You must be willing to start again: experimental code isn't experimental if you're not prepared to throw it away.

(2) Whenever you start coding, you must have a firm idea of what you're trying to implement. There's no point starting to code to a specification that is vague and missing crucial details. That doesn't mean that your specification must be complete and polished, but it does mean that you shouldn't start coding a method until at least you have its own specification written.

(3) You must write down the specification of a method and not just imagine it; it is too easy to delude yourself. Try to write it on paper and mull it over before you start any coding. It's tempting to sit in front of an editor, write some specification as comments, and then start coding around them, but this tends not to be nearly as effective.

Designing Tests

It can be difficult to come up with a good test suite. You would like to test a variety of "interesting" graphs, but what are interesting graphs? One possible approach is a "0, 1, 2" case analysis: test scripts with 0, 1, and 2 graphs are interesting; graphs with 0, 1, and 2 nodes and 0, 1, and 2 edges are interesting. For each method, 0, 1, and 2 parameters and 0, 1, and 2 results are interesting; for example: `addEdge()` on nodes that currently have 0, 1, and 2 children; `listChildren()` on nodes with 0, 1, and 2 children; etc. This approach, while certainly not required, can give a good way to structure your tests to cover many important cases without having too much redundancy. Use EclEmma to measure your coverage and gain even more confidence in your tests.

Abstraction function, representation invariant, and checkRep()

Include an abstraction function, a representation invariant, and a private `checkRep()` method in all new classes you create that represent an ADT. If a class does not represent an ADT, place a comment that explicitly says so where the AF and RI would normally go. (For example, classes that contain only static methods and are never constructed usually do not represent an ADT — though you are unlikely to write any such classes for Homework 4.) Please come to office hours if you feel unsure about what counts as an ADT and what doesn't.

Be conscious of how certain operations in `checkRep()`, particularly iterating over a large dataset, may affect the "big-O" runtime of your methods. If your program suffers performance problems in Homework 4 or 5, `checkRep()` is a good place to start looking for problems.

It is hard to balance the utility of the `checkRep()` method with how expensive it may be to run. A good approach is to call `checkRep()` as much as possible (generally at the beginning and end of every method), but to **disable the `checkRep()` method when you turn in your code so that our tests don't timeout**. A good way to do this is to have a static final constant variable that is checked in your `checkRep()` such that it only runs when the constant variable is set.

A warning about equals() and hashCode()

You may find it useful to define a class or classes that implement method `equals()`. If you do that, *be sure* to also provide a consistent definition of method `hashCode()`, otherwise your objects may behave strangely if used in containers like HashMaps. There is a good discussion of the issues involved in *Effective Java* (item 10 in the 3rd edition). Eclipse will generate an `equals()` and `hashCode()` for you, but if you use it, be sure that it actually fits your needs.

A warning about using generics

You are permitted **but discouraged** from implementing generic classes on this assignment (that will come later). A generic class is one defined as something like:

```
public class Graph<N,E> {...}
```

where a client could then construct a Graph<String,String>, Graph<City,Road>, etc.

If you choose to write a generic class, be aware that Eclipse's built-in compiler sometimes handles generics differently from javac, the standard command-line compiler.

This warning is only directed at students writing their own generic classes. Simply using, say, a List<String> or Comparator<Foo> (as you have been doing all along) should be fine.

What to Submit

Push to git the following files. Don't forget to submit on Submitty!

- `answers/hw4_problem1.pdf`
- `answers/hw4_problem2.pdf`
- `answers/hw4_problem3.pdf`
- `answers/hw4_reflection.pdf`
- `answers/hw4_collaboration.pdf`
- `src/main/java/hw4/*.java` *[Java classes for your graph implementation]*
- `src/test/java/hw4/*Test.java` *[JUnit test classes you create]*

Errata

Check the Submitty Discussion forum regularly. Updates to this homework will be posted there.