6.170 Laboratory in Software Engineering
Fall 2005
Problem Set 4: Design an Abstract Data Type
Due: **Thursday**, October 13, 2005 at 1:00pm

Contents:

- Purpose
- Background
- Exercises
- Design Problem
  - Problem 1
  - Problem 2
  - Problem 3
  - Problem 4
- Mechanics
- Hints
- Expectations
- Errata

We recommend that you read the entire problem set before you begin work.

Purpose

The purpose of this problem set is to improve your understanding of *Abstract Data Types* (ADTs). You will design, implement and test a *directed labeled multi-graph*, and, in so doing, will gain a better appreciation for the role that fundamental tools such as specifications, object models and the rep invariant play in the design of a module.

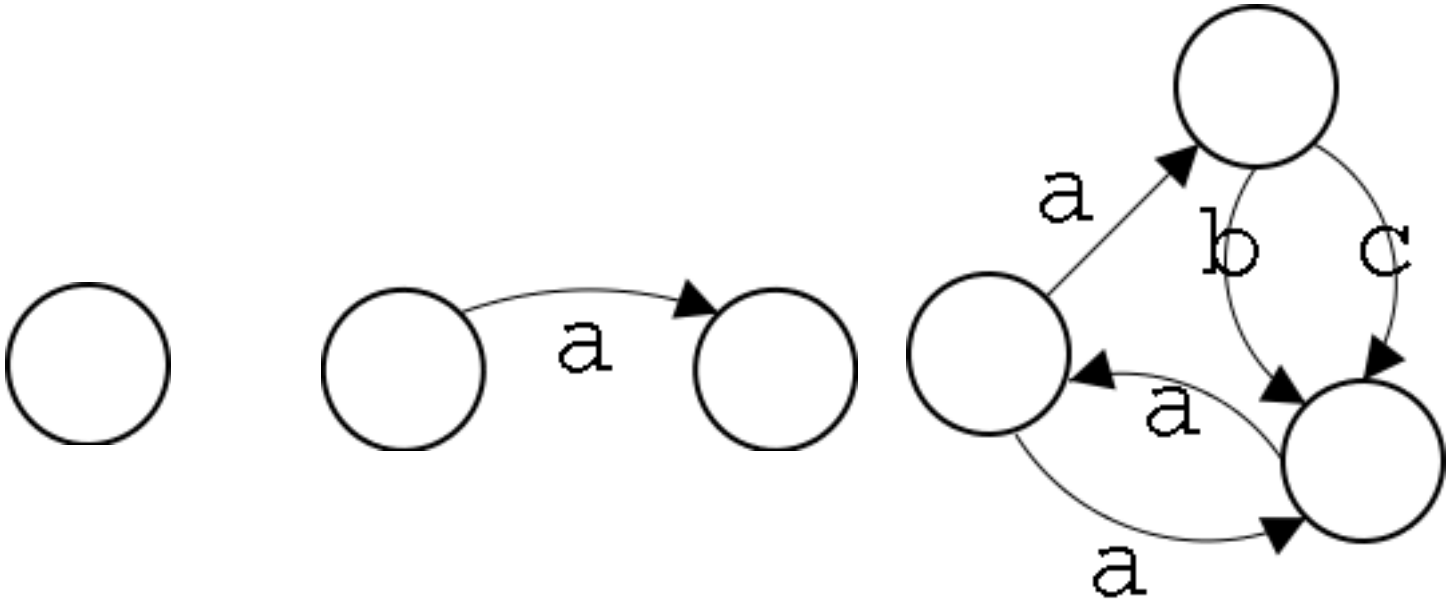
Background

Designing an abstract type involves determining what services it should provide and what their behavior should be -- that is, writing a specification. Implementing an abstract type involves choosing a representation and algorithms, and embodying them in code. The focus of this exercise is the design, so you should expect to spend a considerable amount of time on it. Note the distribution of points among each problem. We advise you to distribute your time accordingly.

Graphs

The abstract type that you'll design is a *directed labeled multi-graph*. A graph is a collection of nodes with edges between them. Every edge connects one node (the source) to one other node (the target). There can be nodes without edges but no edges without nodes. A node may be connected to itself. In a multi-graph, there can be zero, one or more edges between a pair of nodes. Every edge has a label. Distinct edges may have the same label.

Some examples of directed labeled multi-graphs are depicted below.



a graph with 1 node

a graph with 2 nodes and 1 edge

a graph with 3 nodes and 5 edges

Here are some examples of applications of directed labeled multi-graphs.

- **A compiler may represent the control flow of a program as a graph whose nodes are points in the program, and whose edges are program statements. The graph would be used for analyses, such as propagating the values of constants, and for transformations, such as hoisting a statement out of a loop.**
- **A website design tool may represent a website as a graph whose nodes are documents and whose edges are links. The tool may examine the site for connectivity, find broken links, update all documents when a document is moved, and so on.**

- **A curriculum design tool may use a graph to show prerequisite relationships between courses, to find inconsistencies and determine feasible programs of study.**
- **A program for generating driving directions may use a graph to represent a street map, and compute a shortest path to find directions from one point to another.**
- **A Java compiler may use a graph to represent dependences between source code files, in order to determine a reasonable order of compilation.**

Exercises

Approximate time to completion: 15 to 30 minutes

Answer the following questions in a file named `exercises.txt` and put it in the `doc/` directory.

1. **The source code for `TreeSet.java` is provided to you under the `src/java/util/` directory. Read this source code, along with the Java specifications for `TreeSet`, and write the abstraction function and representation invariant for that class.**

   Note: although you will probably not need them, you will also find the source code for `AbstractSet` and `AbstractCollection` in your source directory.

2. **Below are signatures for various methods/constructors. From the information**

available (for example, the return type), which of these methods/constructors could possibly expose the representation? Give a brief explanation in each case.

(a) `public int solveEquations(int x, int y, int z)`

(b) `public String[] decode(boolean slowly)`

(c) `private Date myBirthday()`

(d) `public String toString()`

(e) `public Iterator elements()`

(f) `public Deck(List cards)`

Design Problem

Your task is to design, implement and test an abstract data type for a directed labeled multi-graph.

This datatype will be used to find directions on the Boston subway. In this particular application, the nodes will represent stations and the edges will represent track segments. Your design and implementation should, however, be *polymorphic*. This means that the node type should be generic; it should be possible to use your graph design and implementation (possibly with a few small modifications) in different applications, where nodes and edges represent other things. (Note: we do

not require your ADT to support Java 1.5 parameterization, but that could be a nice feature.)

Your graph implementation must be efficient. This means it should perform reasonably for medium-sized graphs, of thousands (but not millions) of nodes and edges. You should *not* assume that the graph will be *sparse* (that is, containing very few edges compared to nodes) or *dense* (that is, with most node pairs connected). The asymptotic running time of your implementation is what matters. However, don't make your code less general or harder to read for the sake of small efficiency gains. (See Item 37, 'Optimize Judiciously', in Joshua Bloch's *Effective Java*.)

Problem 1: Graph Specification

Approximate time to completion: one hour

Design an abstract data type for a directed labeled multi-graph.

You should hand in the following artifacts:

1. **An object model of the problem domain (Turn in as `doc/pom.gif` or `doc/pom.png`)**
2. **A design rationale (`doc/design.txt`), which includes:**
   - **a brief overview of your design;**
   - **a variety of alternative designs that you considered but rejected, with explanations of why they were rejected;**

- any assumptions you made when designing your ADT.

3. A specification for each public class or interface in your design (placed in the directory `src/ps4/graph/`), containing:
   - an overview paragraph explaining in abstract terms what objects are represented by that class or interface, and whether they are mutable or not;
   - a specification of each non-private method.

The specifications should be in the source code files. You should not turn in the HTML files generated by Javadoc, but you are responsible for making sure that Javadoc can correctly generate HTML files from your source code.

NOTE: Please make sure your GIF or PNG picture files are added to CVS in binary mode. To ensure new files are added in binary mode, go to the menu on top and click on Window. Then click on Preferences..., expand the Team option on the left, then click on the CVS option under Team. Make sure you check the checkbox that says *Treat all new files as binary*. You can then click OK to close the window.

Running `validate6170 ps4` will check to make sure your files are added in the right mode to CVS.

Problem 2: Graph Implementation

Approximate time to completion: two hours

Implement the graph ADT you designed in Problem 1.

You should hand in the following artifacts:

1. **A overview of the implementation you chose (as distinguished from your design), along with a brief rationale justifying your implementation decisions in comparison to alternative implementations. Include rough estimates of the time and space required by your implementation. (Turn this in as `doc/implementation.txt`.)**
2. **A representation invariant and abstraction function. You can use either the 6.170 specification notation, or state these informally. If you state them informally, you will have to be extra careful to be precise. (Turn in as `doc/ri_af.txt`.)**
3. **A code object model representing your specific implementation of a graph. (Turn in as `doc/com.gif` or `doc/com.png`)**
4. **Well-commented source code (placed in the directory `src/ps4/graph/`).**

Problem 3: Testing Graph

Approximate time to completion: one hour

Test your graph ADT.

You should hand in the following elements:

1. **A small collection of plausible JUnit test cases (placed in the directory `src/ps4/tests/`);**
2. **A brief explanation of why the tests you chose to implement increase your confidence on the code you wrote (turned in as `doc/tests.txt`);**

Problem 4: Using Graph

Approximate time to completion: two hours

Write a program named `MetroDirections.java` that generates directions for the Boston subway system. You should implement the `getDirections(String fileName, String origin, String destination)` method which takes an input file name and two station names and returns a String containing directions from the origin to the destination. The string should list stations in successive order, separated by the subway line to be taken between them.

We provide you with a simple text file (`input/bostonmetro.txt`) that describes the Boston subway system, along with a basic parser (`src/ps4/directions/MetroMapParser.java`). You can use (or not use) this code in any way you like; you should be able to just insert calls to your ADT methods to construct the graph from the file. Depending on the design of your graph, you may need to modify the code more or less; you may even require more than one pass through the file. If you use this code, you should clearly document any modifications you make. Furthermore, your code

should be able to handle different input files in the same format.

The format of the output should be:

```
Station1-(Line12)-Station2-(Line23)-Station3-...-StationN-1-(LineN-1N)-StationN
```

where StationX is the name of a station (e.g. `Kendall`), and LineXY is the name of a line (e.g. `GreenB`). For example: `getDirections("input/bostonmetro.txt", "Kendall", "Haymarket")` should return:

```
Kendall-(Red)-Charles/MGH-(Red)-ParkStreet-(Green)-GovernmentCenter-(Green)-Haymarket
```

A simple breadth-first search is adequate. You're not expected to make this program realistic. But we would like you to think a bit about what complications would arise if you were to do so.

Be sure to run `validate6170 ps4` when you're finished. You should hand in the following artifacts:

1. Well-commented source code (placed in the directory `src/ps4/directions/`);
2. Sample output of running `getDirections()` on at least four reasonably comprehensive test cases (turned in as the file `doc/samples.txt`). For debugging purposes, we have included a skeleton JUnit test file in `MetroDirectionsTest.java`. Your code should pass the test there, and we encourage you to append your own tests to the file.
3. Some comments about what complications would arise in a more realistic design of a directions

finder for the Boston subway, focusing on the structure of the subway and the directions (turned in as the file `doc/comments.txt`).

Mechanics

In your repository

In your CVS repository you will find the following files:

- `src/java/util/*.java` : the source code for `TreeSet`.
- `src/ps4/graph/Graph.java` : a skeleton file for your Graph ADT.
- `src/ps4/directions/*.java` : the code we give you for your directions finder.
- `src/ps4/tests/*.java` : skeleton files for JUnit tests.
- `input/bostonmetro.txt` : a description of the Boston subway system.
- `build.xml` : an ant script that generates the Javadoc files for all source files located at `src/ps4`.

Checking in

You should submit the following elements:

Text files

- `doc/exercises.txt`
- `doc/design.txt`
- `doc/implementation.txt`
- `doc/ri_af.txt`
- `doc/tests.txt`
- `doc/samples.txt`
- `doc/comments.txt`

Object models

- `doc/pom.gif` or `doc/pom.png`
- `doc/com.gif` or `doc/com.png`

Source files in the following directories

- `src/ps4/graph/`
- `src/ps4/tests/`
- `src/ps4/directions/`

Note: You should run `validate6170 ps4` on Athena to verify that all the required files are in CVS and that your Java files are compilable.

Hints

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 careful judgment, and think carefully about how decisions you make interfere with each other.

- **will the graph be mutable or immutable? will it be possible to change the label of an edge?**
- **will edge labels be strings or generic objects? if objects, will it be OK to use a node or an edge as a label? or even a graph?**
- **will nodes be required only to satisfy the interface of java.lang.Object? or will you design a Java interface for nodes?**

- **will the graph be implemented as a single class, or will there be a separate Java interface for the Graph specification, and a class for the implementation?**
- **will edges be objects in their own right? will they be visible to a client of the abstract type?**
- **will it be possible to find the successor of a node from the node alone, or will the graph be needed too? can a node belong to multiple graphs?**
- **what kind of iterators will the type provide?**
- **should path-finding operations be included as methods of the graph, or should they be implemented in client code on top of the graph?**
- **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?**

You may find the `Integer.parseInt(String s)` method to be of use in converting `String`'s into `int`'s.

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 will be 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 two things in mind. First, you must be willing to start again: experimental code isn't experimental if you're not prepared to throw it away. Second, 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. Third, you *must* write down the specification of a method and not just imagine it; it's 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 so effective.

Make good use of your TA. Take your specification to office hours before going to the lab and get some feedback on your design and style. This is likely to save you a lot of time!

Expectations

What you need to know before you start:

You should be familiar with the basic concepts of data abstraction, and what comprises a good abstract type; with the notions of representation invariant and abstraction function; how to express structure as an object model; how to write specifications of methods; and you should have by now some experience writing Java code.

What you should expect to learn:

You will learn about the challenges of designing a useful data type. You should also expect to gain experience with the process of designing flexible and robust software.

How you will be evaluated:

- **15 points for the exercises.**
- **35 points for the graph design and its justification.**
- **25 points for the implementation: the choice of rep invariant and abstraction function, and for writing clean and well-organized code.**
- **10 points for a reasonable selection of unit test cases for the graph datatype.**
- **15 points for the directions finder.**

Errata

There are no known problems with this problem set.