Problem Set 3
Implementing Data Abstractions

Out: 11 September
Due: **Monday, 18 September**
(beginning of class)

Collaboration Policy. For this problem set, you may either work alone and turn in a problem set with just your name on it, or work with one other student in your section of your choice. If you worked with a partner on PS2, you may not work with the same person for PS3. If you work with a partner, you and your partner should turn in one assignment with both of your names on it.

Regardless of whether you work alone or with a partner, you are encouraged to discuss this assignment with other students in the class and ask and provide help in useful ways. You may consult any outside resources you wish including books, papers, web sites and people. If you use resources other than the class materials, indicate what you used along with your answer.

> **Reading:** Before beginning this assignment, you should finish reading Chapter 5 and read Section 10.4 (on Testing Data Abstractions).

Purpose

- Gain experience implementing a data abstraction.
- Learn to use abstraction functions and rep invariants to reason about data abstractions.
- Reason about design trade-offs in implementing data abstractions.

Reasoning About Data Abstractions

First, we will consider a implementing the `Poly` abstraction from Chapter 5 (specified in Figure 5.4). Suppose we implement `Poly` using a java.util.Vector. The `Vector` type allows us to represent an ordered collection of Objects. The objects we will store in our vector are records of <term, coefficient>. Here is the representation:

```
import java.util.Vector;

class TermRecord { // OVERVIEW: Record type
   int power;
   int coeff;
   TermRecord (int p_coeff, int p_power);
}

public class Poly {
   // OVERVIEW: Polys are immutable polynomials with integer
   //      coefficients. A typical Poly is:
   //          c_0 + c_1*x + c_2 * x^2 + ...

   private Vector<TermRecord> terms;
   ...
}
```

Suppose this is the implementation of `degree`:

```
   public int degree () {
      // EFFECTS: Returns the degree of this, i.e., the largest exponent
      //    with a non-zero coefficient.  Returns 0 if this is the zero Poly.
      return terms.lastElement ().power;
    }
```

> **1.** What abstraction function and rep invariant are needed to make the implementation of `degree` above satisfy its specification?

In an alternate implementation with the same rep (that is, the abstraction function and rep invariant may be different), suppose this is the implementation of `coeff`:

```
public int coeff (int d) {
   // EFFECTS: Returns the coefficient of the term of this whose
   //     exponent is d.

   int res = 0;
   for (TermRecord r : terms) {
     if (r.power == d) { res += r.coeff; }
   }

   return res;
}
```

2. What rep invariant would make the implementation of `coeff` above correctly satisfy its specification?

3. Explain how a stronger rep invariant would make it possible to implement `coeff` more efficiently.

Implementing StringGraph

Here is the specification of a `StringGraph` datatype. It is similar to the `DirectedGraph` datatype, except for instead of being a generic type with the node type as a type parameter, it uses String for the node type. (In the later questions, you will implement the generic `DirectedGraph` datatype.)

```
public class StringGraph
  OVERVIEW: A StringGraph is a directed graph  where
      V is a set of Strings, and E is a set of edges.
      Each edge is a pair (v1, v2), representing an edge
      from v1 to v2 in G.  A typical StringGraph is
       < {v1, v2, ..., vn} , { { v_a1, v_b1 }, { v_a2, v_b2 } , ... } >

  public StringGraph()
    EFFECTS: Creates a new, empty StringGraph: < {}, {} >

  public void addNode(String s) throws DuplicateException
     MODIFIES: this
     EFFECTS: If s is the name of a node in this, throws
       DuplicateException. Otherwise, adds
       s to the nodes in this, with no adjacent nodes:
           G_post = < V_pre U { s }, E_pre >

  public void addEdge(String s, String t) throws NoNodeException, DuplicateException
     MODIFIES: this
     EFFECTS: If s and t are not nodes in
        this, throws NoNodeException.  If there is already an edge
        between s and t, throws DuplicateEdgeException.
        Otherwise, adds an edge between s and t to this:
           G_post = < V_pre, E_pre U  >

  public Set<String> getAdjacent(T s) throws NoNodeException
     EFFECTS: If s is not a node in this, throws NoNodeException.
        Otherwise, returns a set containing the nodes adjacent to s
        That is, returns the set of nodes
            { e | <s, e> is in E }
```

```
public String toString()
   EFFECTS: Returns a human-readable string representation of this.
```

There are many possible representations for the `StringGraph`. Here are three possibilities:

a

```
Vector<String> nodes;
boolean [][] edges;
```

b

```
Set<String> nodes;
Set<Edge> edges;
```

where `Edge` is a record type containing two `String` values:

```
class Edge {
   String a, b;
   Edge (String p_a, String p_b);
}
```

c

```
Set<NodeRecord> rep;
```

where `NodeRecord` is a record type that records a String and an associated set of Strings:

```
class NodeRecord {
   String key;
   Set<String> values;
}
```

4. For each representation choice (a, b, and c), provide an abstraction function and rep invariant.

5. Which representation choice would make implementing `addNode` most difficult? Explain why.

6. Which representation choice would enable the most efficient `getAdjacent` implementation? Explain why.

7. Implement the `StringGraph` datatype specified above. You may use any of the datatypes from PS2 you want *except* the `DirectedGraph` datatype (note that the specification suggests using the `NoNodeException`, `DuplicateException`, and `Set` provided datatypes). Your implementation may use any representation you want (including the ones describe above, but not limited to those choices). Your implementation should clearly document its abstraction function and rep invariant.

8. Describe a testing strategy for your `StringGraph` datatype. Include all the code you developed for testing in your answer.

9. Consider adding a `removeNode` method to the `StringGraph` datatype that removes a node from a graph. Write a declarative specification for the `removeNode` method. Consider carefully what should happen with the edges of the graph when a node it removed, and make sure your specification is total.

10. For this question you have a choice, either do choice 1 or choice 2:

- Choice 1: Implement the `removeNode` method you specified in question 8.
- Choice 2: Implement the generic `DirectedGraph` datatype (as specified in Problem Set 2) that generalizes the node type to be any object type instead of `String`.

Turn-in Checklist: You should turn in your answers to questions 1-10 on paper at the beginning of class on Monday, 18 September. Also, submit your `StringGraph.java` code electronically by email to *evans@cs.virginia.edu*.