Implementing Data Abstractions

The most important decision in implementing a data abstraction, is selecting the concrete representation and understanding the mapping between that representation and the abstract values.

Abstraction Function

The Abstraction Function maps a concrete state to an abstract state:

\[ AF: C \rightarrow A \]

It is a function from concrete representation to the abstract notation introduced in overview specification.

Representation Invariant

The Representation Invariant expresses properties all objects of the ADT must satisfy. It is a function from concrete representation to a Boolean:

\[ I: C \rightarrow \text{boolean} \]

To check correctness we assume all objects passed in to a procedure satisfy the invariant and prove all objects satisfy the invariant before leaving the implementation code.

/**
 * OVERVIEW: A StringStack represents a last-in-first-out stack where all elements are Strings.
 * A typical stack is \([ e_{n-1}, e_{n-2}, ..., e_1, e_0 ]\) where \(e_{n-1}\) is the top of the stack.
 */
public class StringStack {
    // Rep:
    private List<String> rep;

    // Abstraction function:

    // Rep Invariant:
Here is the specification for an undirected graph datatype. It has some similarities to the StringGraph (directed graph) datatype from ps3, but some differences also.

public class Graph
  // OVERVIEW: A Graph is a mutable type that represents an undirected graph. It consists of
  // nodes that are named by Strings, and edges that connect a pair of nodes.
  // A typical Graph is: < Nodes, Edges > where
  // Nodes = { n_1, n_2, ..., n_m }
  // Edges = { {a_1, b_1}, ..., {a_n, b_n} } (the elements of Edges are unordered sets).

public Graph()
  // EFFECTS: Initializes this to a graph with no nodes or edges: < {}, {} >.

  // Mutators
public void addNode (String name) throws DuplicateException
  // MODIFIES: this
  // EFFECTS: If name is in Nodes, throws DuplicateException.
  //  Otherwise, adds a node named name to this:
  //   this_post = < Nodes_pre U { name }, Edges_pre >

public void addEdge (String s, String t)
  throws NoNodeException, DuplicateException
  // MODIFIES: this
  // EFFECTS: If s and t are not names of 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:
  //  this_post = < Nodes_pre, Edges_pre U {s, t} >

  // Observers
public boolean hasNode (String node)
  // EFFECTS: Returns true iff node is a node in this.

public Set<String> getNeighbors (String node)
  // REQUIRES: node is a node in this
  // EFFECTS: Returns the set consisting of all nodes in this
  // that are directly connected to node:
  //  { n | {node, n} is in this.edges }

1. Select a representation. Consider carefully several different possible representations, and what their advantages and disadvantages will be.
2. Determine the rep invariant and abstraction function
3. Implement DirectedGraph(), addNode and hasNode, addEdge and getNeighbors.