cs2220: Engineering Software

Class 10: Generic Datatypes

Fall 2010
University of Virginia
David Evans

Exam 1

- Out at end of class today, **due Tuesday at beginning of class**
  - I will answer questions about PS3 (including anything in PS3 comments) while the exam is out (including Monday's office hours)
- **Work alone**
- **Open resources**: use any resources you want, but cite anything that is not part of the course materials

PS3: Available for pick-up by Monday.

Reality Check

- Writing abstraction functions, rep invariants, testing code thoroughly, reasoning about correctness, etc. for a big program is a ridiculous amount of work!
- Does anyone really do this?
  - Yes (and a lot more), but usually only when its really important to get things right:
- Cost per line of code:
  - Small, unimportant projects: $1-5/line
  - WindowsNT: about $100/line
  - FAA's Automation System (1982-1994): $900/line

StringGraph, IntegerGraph, etc.

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), …

public class IntegerGraph
OVERVIEW: A IntegerGraph is a directed graph where V is a set of …

public class CityGraph

public class URLGraph
OVERVIEW: A URLGraph is a directed graph where V is a set of URL objects, 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 IntegerGraph is < {v1, v2, …, vn} , { (v_a1, v_b1), (v_a2, v_b2), … } > where each ai and bi is in [1, n].
public void addNode(URL s) throws DuplicateException;
public void addEdge(URL s, URL t) throws NoNodeException, DuplicateException
…

Type Parameters

- We want to have Graphs with many different types of nodes
- How should we declare the **Graph** methods?

```
public void addNode(? s)
public void addEdge(? s, ? t)
public Set<?> getAdjacent()
```

We don't want just one Graph datatype.
We want different Graphs for different node types.

Generic Datatype

```
public class StringGraph {
  …
  public void addNode(String s) …
  public void addEdge(String s, String t) …
  public Set< String> getAdjacent() …
  …
}
```

Graph<String>

```
public class Graph<T> {
  …
  public void addNode(T s) …
  public void addEdge(T s, T t) …
  public Set<T> getAdjacent() …
  …
}
```

Note: Java did not support generic datatypes until version 1.5 (this is why the book doesn't use them)

Specifying the Generic Graph

```
public class Graph<T>
  OVERVIEW: A Graph is a directed graph where V is a set of
   T objects, 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 Graph is
    < {v1, v2, ..., vn} , { (v_a1, v_b1), (v_a2, v_b2), ... } > where each ai
   and bi is in [1, n].

  public void addNode(T s) throws DuplicateException
     // MODIFIES: this
     // EFFECTS:
```

```
public class 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 >
```

(same is s.equals(t))

Implementing a Generic Graph

```
// Rep C
public class StringGraph {
 class NodeRecord implements Comparable<NodeRecord> {
   String key;
   Set<String> values;
   ...
 }

 private Set<NodeRecord> rep;
```

<T> — Set<T> — int compareTo(NodeRecord n)

```
public class Graph<T extends Comparable<T>>  {
  class NodeRecord implements Comparable<NodeRecord> {
    T key;
    Set<T> values;
    ...
  }
  private Set<NodeRecord> rep;
```

NodeRecord (for StringGraph)

public class Graph<T extends Comparable<T>>{

```
class NodeRecord implements Comparable<NodeRecord> {
   String key;
   Set<String> values;

   NodeRecord (String p_key) {
     key = p_key;
     values = new TreeSet<String>();
   }
   public int compareTo(NodeRecord n) {
     // This assertion should be guaranteed by the rep invariant
     assert (!key.equals(n.key) || (n.values == values));
     return key.compareTo(n.key);
   }
   public boolean matches(String p_key) {
     return key.equals(p_key);
   }

   public void addEdge(String p_edge) throws DuplicateException { ... }
}
```

//Rep Inv key != null — this == n — (String → T)

Its okay that the rep isn't private, since this is an **inner class**: defined inside **StringGraph** implementation, not visible to clients.

We need to implement compareTo to satisfy the Comparable<NodeRecord> interface, required by TreeSet.

Generic NodeRecord

```
class NodeRecord implements Comparable<NodeRecord> {
  T key;
  Set<T> values;
  NodeRecord (T p_key) {
    key = p_key;
    values = new TreeSet<T>();
  }

  public int compareTo(NodeRecord n) {
    // This assertion should be guaranteed by the rep invariant
    assert (!key.equals(n.key) || (n.values == values));
    return key.compareTo(n.key);
  }

  public boolean matches(T p_key) {
    return key.equals(p_key);
  }

  public void addEdge(T p_edge) throws D
  {
    if (values.contains(p_edge)) throw new
    values.add(p_edge);
  }
}
```

Abstraction Function:
  Nodes = { el.key | el is an element in rep }
  Edges = { (el.key, value) | el is an element in rep and value is an element in el.values }
Rep Invariant:
  rep != null
  elements of rep are not null
  forall elements e in rep:
    forall elements f in e.values: f = el.key for some element el in rep

f.equals(el.key)

```
public class Graph<T extends Comparable<T>>  {
  class NodeRecord implements Comparable<NodeRecord> { ... }
  private Set<NodeRecord> rep;

  public Graph() {
    rep = new TreeSet<NodeRecord>();
  }

  private NodeRecord getNode(T s) throws NoNodeException {
    // EFFECTS: If s is a node in this returns the corresponding NodeRecord.  If not, throws NoNodeException.
    for (NodeRecord r : rep) {
      if (r.matches(s)) {
        return r;
      }
    }
    throw new NoNodeException(s.toString());
  }
public void addNode(T s) throws DuplicateException {
    try {
    getNode(s);
      throw new DuplicateException(s.toString());
    } catch (NoNodeException e) {
      // No exception means its not a duplicate
      rep.add(new NodeRecord(s));
    }
  }
```

new Set<T> X Set is "abstract class" — returns null — if (getNode(s) != null)

rep: Set<NodeRecord> rep;

class NodeRecord { T key; Set<T> values; }

```
public void addEdge(T s, T t) throws NoNodeException, DuplicateException {
    NodeRecord srec = getNode(s);
    getNode(t); // only to get exception if t doesn't exist
    srec.addEdge(t); // DuplicateException will be rethrown
  }

public Set<T> getAdjacent(T s) throws NoNodeException {
    return new TreeSet<T>(getNode(s).values);
  }
```

Set<T>

Why do we need to copy values?

Typical graph is <V, E>
V = { n_1, ... n_n }
E = { (n_a, n_b), ... ( ) }

```
public String toString() {
     String res = "<";
     boolean firstone = true;
     for (NodeRecord r : rep) {
       if (firstone) { firstone = false; } else { res += ", “; }
       res += r.key;
     }
     res += "; { ";
     firstone = true;
     for (NodeRecord r : rep) {
       for (T e : r.values) {
         if (firstone) { firstone = false; } else { res += ", “; }
           res += "(" + r.key + ", " + e + ")";
       }
     }
     res += " } >";
     return res;
   }
}
```

Abstraction Function:
Nodes = { el.key | el is an element in rep }
Edges = { (el.key, value) | el is an element in rep and value is an element in el.values }

class City implements Comparable {

Graph<City> citygraph;

Object
String toString()

T.toString()

Charge

Exam 1 is due Tuesday

Next week: Subtyping

PS4: Designing with Data Abstractions and Subtyping