Generic Method Pattern

From `http://net.datastructures.net/`

1 DFS generic class

```
1  package net.datastructures;
2  import java.util.Iterator;
3
4  /** Generic DFS traversal of a graph using the template method pattern.
5    * Parameterized types:
6    * V, the type for the elements stored at vertices
7    * E, the type for the elements stored at edges
8    * I, the type for the information object passed to the execute method
9    * R, the type for the result object returned by the DFS
10   */
11 public class DFS<V, E, I, R> {
12
13   protected Graph<V, E> graph; // The graph being traversed
14
15   protected Vertex<V> start; // The start vertex for the DFS
16
17   protected I info; // Information object passed to DFS
18
19   protected R visitResult; // The result of a recursive traversal call
20
21   protected static Object STATUS = new Object(); // The status attribute
22
23   protected static Object VISITED = new Object(); // Visited value
24
25   protected static Object UNVISITED = new Object(); // Unvisited value
```

```
26    /** Execute a depth first search traversal on graph g, starting
27      * from a start vertex s, passing in an information object (in) */
28    public R execute(Graph<V, E> g, Vertex<V> s, I in) {
29      graph = g;
30      start = s;
31      info = in;
32      for(Vertex<V> v: graph.vertices()) unVisit(v); // mark vertices as unvisited
33      for(Edge<E> e: graph.edges()) unVisit(e); // mark edges as unvisited
34      setup(); // perform any necessary setup prior to DFS traversal
35      return finalResult(dfsTraversal(start));
36    }
37
38    /** Recursive template method for a generic DFS traversal. */
39    protected R dfsTraversal(Vertex<V> v) {
40      initResult();
41      if (!isDone())
42        startVisit(v);
43      if (!isDone()) {
44        visit(v);
45        for (Edge<E> e: graph.incidentEdges(v)) {
46          if (!isVisited(e)) {
47            // found an unexplored edge, explore it
48            visit(e);
49            Vertex<V> w = graph.opposite(v, e);
50            if (!isVisited(w)) {
51              // w is unexplored, this is a discovery edge
52              traverseDiscovery(e, v);
53              if (isDone()) break;
54              visitResult = dfsTraversal(w); // get result from DFS−tree child
55              if (isDone()) break;
56            }
57            else {
58              // w is explored, this is a back edge
59              traverseBack(e, v);
60              if (isDone()) break;
61            }
62          }
63        }
64      }
65      if(!isDone())
66        finishVisit(v);
67      return result();
68    }
```

```
69    /** Mark a position (vertex or edge) as visited. */
70    protected void visit(DecorablePosition<?> p) {
71      p.put(STATUS, VISITED);
72    }
73
74    /** Mark a position (vertex or edge) as unvisited. */
75    protected void unVisit(DecorablePosition<?> p) {
76      p.put(STATUS, UNVISITED);
77    }
78
79    /** Test if a position (vertex or edge) has been visited. */
80    protected boolean isVisited(DecorablePosition<?> p) {
81      return (p.get(STATUS) == VISITED);
82    }
83
84    // Auxiliary methods (all initially null) for specializing a generic DFS
85    /** Setup method that is called prior to the DFS execution. */
86    protected void setup() {}
87
88    /** Initializes result (called first, once per vertex visited). */
89    protected void initResult() {}
90
91    /** Called when we encounter a vertex (v). */
92    protected void startVisit(Vertex<V> v) {}
93
94    /** Called after we finish the visit for a vertex (v). */
95    protected void finishVisit(Vertex<V> v) {}
96
97    /** Called when we traverse a discovery edge (e) from a vertex (from). */
98    protected void traverseDiscovery(Edge<E> e, Vertex<V> from) {}
99
100   /** Called when we traverse a back edge (e) from a vertex (from). */
101   protected void traverseBack(Edge<E> e, Vertex<V> from) {}
102
103   /** Determines whether the traversal is done early. */
104   protected boolean isDone() { return false; /* default value */ }
105
106   /** Returns a result of a visit (if needed). */
107   protected R result() { return null; /* default value */ }
108
109   /** Returns the final result of the DFS execute method. */
110   protected R finalResult(R r) { return r; /* default value */ }
111 }
```

2 Application: Test if graph connected

```
1  package net.datastructures;
2  import java.util.Iterator;
3  import java.lang.Boolean;
4
5  /** This class specializes DFS to determine whether the graph is connected.
6   * Input to the execute method are the graph itself, a start vertex, the info
7   * parameter is unused, and the return type is a Boolean.
8   */
9  public class ConnectivityDFS<V, E> extends DFS <V, E, Object, Boolean> {
10   protected int reached;
11
12   protected void setup() { reached = 0; }
13
14   protected void startVisit(Vertex<V> v) { reached++; }
15
16   protected Boolean finalResult(Boolean dfsResult) {
17     return new Boolean(reached == graph.numVertices());
18   }
19 }
```

3 Application: Find a path between two nodes in a graph

```
1  package net.datastructures;
2
3  /** Class specializing DFS to find a path between a start vertex and a target
4    * vertex. It assumes the target vertex is passed as the info object to the
5    * execute method. It returns an iterable list of the vertices and edges
6    * comprising the path from start to info. The returned path is empty if
7    * info is unreachable from start. */
8  public class FindPathDFS<V, E>
9        extends DFS<V, E, Vertex<V>, Iterable<Position>> {
10   protected PositionList<Position> path;
11   protected boolean done;
12
13   /** Setup method to initialize the path. */
14   public void setup() {
15     path = new NodePositionList<Position>();
16     done = false;
17   }
18
19   protected void startVisit(Vertex<V> v) {
20     path.addLast(v); // add vertex v to path
21     if (v == info)
22       done = true;
23   }
24
25   protected void finishVisit(Vertex<V> v) {
26     path.remove(path.last()); // remove v from path
27     if(!path.isEmpty())  // if v is not the start vertex
28       path.remove(path.last()); // remove discovery edge into v from path
29   }
30
31   protected void traverseDiscovery(Edge<E> e, Vertex<V> from) {
32     path.addLast(e); // add edge e to the path
33   }
34
35   protected boolean isDone() {
36     return done;
37   }
38
39   public Iterable<Position> finalResult(Iterable<Position> r) {
40     return path;
41 }}
```

4 Application: Find a cycle in a graph

```
1  package net.datastructures;
2  import java.util.Iterator;
3
4  /** This class specializes DFS to find a cycle.
5   * Input for the execute method is the graph itself, a start
6   * vertex, the info parameter is unused, and the return type is
7   * an Iterable<Position>.
8   */
9
10 public class FindCycleDFS<V, E>
11   extends DFS<V, E, Object, Iterable<Position>> {
12
13   protected PositionList<Position> cycle; // sequence of edges of the cycle
14   protected boolean done;
15   protected Vertex<V> cycleStart;
16
17   /**
18    * Executes the DFS algorithm.
19    * @param info unused
20    * @return {@link Iterable} collection containing the vertices and
21    * edges of a cycle.
22    */
23
24   public void setup() {
25     cycle = new NodePositionList<Position>();
26     done = false;
27   }
28
29   protected void startVisit(Vertex<V> v) { cycle.addLast(v); }
30
31   protected void finishVisit(Vertex<V> v) {
32     cycle.remove(cycle.last()); // remove v from cycle
33     if (!cycle.isEmpty()) cycle.remove(cycle.last()); // remove edge into v from cycle
34   }
35
36   protected void traverseDiscovery(Edge<E> e, Vertex<V> from) {
37     cycle.addLast(e);
38   }
```

```
39    protected void traverseBack(Edge<E> e, Vertex<V> from) {
40      cycle.addLast(e);  // back edge e creates a cycle
41      cycleStart = graph.opposite(from, e);
42      cycle.addLast(cycleStart); // first vertex completes the cycle
43      done = true;
44    }
45
46    protected boolean isDone() { return done; }
47
48    public Iterable<Position> finalResult(Iterable<Position> r) {
49      // remove the vertices and edges from start to cycleStart
50      if (!cycle.isEmpty()) {
51        for (Position<Position> p: cycle.positions()) {
52          if (p.element() == cycleStart)
53            break;
54          cycle.remove(p); // remove vertex from cycle
55        }
56      }
57      return cycle; // list of the vertices and edges of the cycle
58    }
59  }
```