HEAD:

JDSL: The Data Structures Library In Java

DEK

Making advanced algorithms and data structures a programming reality

Bio

The authors can be contacted at jdsl@cs.brown.edu.

Byline

Roberto Tamassiayz, Michael T. Goodrich,
Luca Vismara, Mark Handy, Galina Shubina,
Robert Cohen, Benoit Hudson, Ryan S. Baker,
Natasha Gelfand, and Ulrik Brandes

**NOTE TO AUTHORS:
PLEASE CHECK ALL URLs AND SPELLINGS OF NAMES CAREFULLY.
THANK YOU.**

Pullquotes

The library was designed so that each algorithm uses data structures only via the interface methods

A good library of algorithms should be able to integrate smoothly with other existing libraries

Once mainly used as number processors to perform fast numerical computations, computers have evolved into information processors for storing, analyzing, searching, transferring, and updating large collections of structured information. For computer programs to perform these tasks effectively, the data they manipulate must be well organized, and the methods for accessing and maintaining those data must be reliable and efficient. In other words, programs need advanced data structures and algorithms. However, implementing advanced data structures and algorithms is not an easy task and presents some risks because of their complexity, proneness to subtle errors, and long development time. Consequently, programmers tend to ignore advanced data structures and algorithms, opting for simple, less efficient ones that are easier to implement and test. Clearly, the development of complex software applications— in particular their rapid prototyping— can benefit from libraries of reliable and efficient data structures and algorithms.

Various libraries are available for C++, including the Standard Template Library (STL; see *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library,* by D.R. Musser and A. Saini, Addison-Wesley, 1996), now part of the C++ Standard; the Library of Efficient Data Structures and Algorithms (LEDA; see *LEDA: A Platform for Combinatorial and Geometric Computing,* by K. Mehlhornand and S. Naher, Cambridge University Press, 1999); and the Generic Graph Component Library (GGCL; see “The Generic Graph Component Library,” by J. Siek, Lie-Quan Lee, and A. Lumsdaine, *DDJ*, September 2000).

The situation with Java is different, however. A small library of data structures and algorithms, usually referred to as “Java Collections” (JC), is included in the Java 2 java.util package (http://java.sun.com/j2se/1.3/). Likewise, there’s the Generic Library for Java (JGL) by ObjectSpace (http://www.objectspace.com/products/voyager/jgl.asp/), which is patterned after STL. Both the JC and JGL provide implementations of basic data structures such as maps, sets, dictionaries, and sequences. JGL also provides a number of template-based algorithms for permuting data. The Graph Foundation Classes for Java (GFC) by IBM’s alphaWorks (http://www.alphaworks.ibm.com/tech/gfc/), a framework for programming with graphs is still in a preliminary stage. The GFG provides more advanced data structures (such as trees and graphs) and some graph drawing algorithms. However, none of these Java libraries provide a coherent framework— capable of accommodating both elementary and advanced data structures and algorithms— required by developers of complex software.

With this in mind, we designed and developed the Data Structures Library in Java (JDSL), a collection of Java interfaces and classes implementing fundamental data structures and algorithms such as:

- Sequences, trees, priority queues, search trees, and hash tables.
- Sorting and searching algorithms.
- Graphs.
- Graph traversals, topological sorting, shortest path, and minimum spanning tree.

The library was designed so that each data structure is specified by an interface and each algorithm uses data structures only via the interface methods. Actual classes need only be specified when objects are instantiated. Programming through interfaces (rather than through actual classes) creates more general code. It lets different implementations of the same interface be used interchangeably, without having to modify the algorithm code. This way, users can choose the most appropriate implementation, in terms of time or space complexity, for the application at hand.

Initial JDSL development began in September 1996 at the Center for Geometric Computing at Brown University and Johns Hopkins University, and culminated with the release of JDSL 1.0 in 1998. A major part of the project in the first year was the experimentation with different models for data structures and algorithms, and the construction of prototypes. A significant reimplementation, documentation, and testing effort was carried out in 1999 in collaboration with Algomagic Technologies (http://www.algomagic.com/) leading to the current JDSL 2.0; this version was officially released in August 2000. The two JDSL releases were accompanied by the publication of the book *Data Structures and Algorithms In Java,* by Michael Goodrich and Roberto Tamassia (John Wiley & Sons, 1998).

JDSL comes with extensive documentation, including detailed Javadoc, overview, tutorial with seven lessons, and several associated research papers. It is available free of charge for noncommercial use (see http://www.cs.brown.edu/cgc/jdsl/). Commercial licenses are also available. Table 1 compares the key features of JC, JGL, GFC, and JDSL, respectively. In our opinion, the main advantages of JDSL are the definition of a large set of data structure APIs in terms of Java interfaces (in particular the tree and graph APIs), availability of reliable and efficient implementations of those APIs, and availability of some fundamental graph algorithms.

A good library of data structures and algorithms should be able to integrate smoothly with other existing libraries. In particular, we have pursued compatibility with the Java Collections. JDSL supplements the JC and is not meant to replace them. No conflicts arise when using data structures from JDSL and JC in the same program. To facilitate the use of JDSL data structures in existing programs, we provided adapter classes to translate a Java Collection into a JDSL container and vice versa, whenever such a translation is applicable.

JDSL Data Organization Concepts

Data structures in JDSL are viewed as containers; that is, as an organized collection of objects (called the "elements" of the container). An element can be stored in many containers at the same time and stored multiple times in the same container. Each JDSL container element is an instance of *java.lang.Object;* this lets containers store heterogeneous elements. JDSL provides two general and implementation-independent ways to access (but not modify) elements stored in a container: individually, by means of accessors; and globally, by means of iterators. An accessor abstracts the notion of membership of an element into a container, hiding the details of the implementation. It provides constant-time access to an element stored in a container, independently from its implementation. Every time an element is inserted in a container, an accessor associated with it is returned. Most operations on JDSL containers take one or more accessors as their operands.

As Figure 1 illustrates, we distinguish between two kinds of containers and accessors:

- Positional containers. Typical examples are sequences, trees, and graphs. In a positional container, some topological relation is established among the placeholders that store the elements (such as the predecessor-successor relation in a sequence), the parent-child relation in a tree, and the incidence relation in a graph. When inserting an element in the container, users decide what the relationship is between the new placeholder and existing ones (in a sequence, for instance, users may decide to insert an element before a given placeholder). A positional container does not change its topology, unless users specifically request a change. The implementation of these containers usually involves linked structures or arrays.

- Positions. The concept of position is an abstraction of the various types of placeholders in the implementation of a positional container (typically the nodes of a linked structure or cells of an array). Each position stores an element. Position implementations can store the following additional information: adjacent positions (that is, the previous and next positions in a sequence, the right and left child and the parent in a binary tree, the list of incident edges in a graph); and consistency information (what container the position is in).

  A position can be directly queried for its element through method *element( ),* which hides the details of where the element is actually stored, be it an instance variable or array cell. Instead, through the positional container, it is possible to replace the element of a position or swap the elements between two positions. As an element moves about in its container (or even from container to container), its position changes. Positions are similar to the concept of items used in LEDA.
- Key-based containers. Typical examples of key-based containers are dictionaries and priority queues. Each element stored in a key-based container has a key associated with it; keys are used as an indexing mechanism for associated elements. Each key of a key-based container is an instance of *java.lang.Object.* Typically, key-based containers are internally implemented using a positional container; for example, a possible implementation of a priority queue uses a binary tree (heap). The details of the internal representation, however, are completely hidden to users. Thus, users have no control over the organization of the positions that store the key/element pairs. It is the key-based container itself that modifies its internal representation based on the keys of the key/element pairs inserted or removed.
- Locators. The key/element pairs stored in a key-based container may change their positions in the underlying positional container due to some internal restructuring; say, after the insertion of a new key/element pair. For example, in the binary tree implementation of a priority queue, the key/element pairs move around the tree to preserve the top-down ordering of the keys, and thus their positions change. Hence, a different, more abstract type of accessor called a "locator" is provided to access a key/element pair stored in a key-based container. Locators hide the complications of dynamically maintaining the implementation-dependent binding between the key/element pairs and their positions in the underlying positional container implementation. A locator can be directly queried for its key and element, and through the key-based container, it is possible to replace the key and element of a locator.

While accessors let users access single elements or key/element pairs in a container, iterators provide a simple mechanism for iteratively listing through a collection of objects. JDSL provides various iterators over the elements, keys, and accessors of a container; see Figure 2. They are similar to the iterators provided by the Java Collections.

All JDSL containers provide methods that return iterators over the entire container (that is, all the nodes of a tree or all the locators of a dictionary). In addition, some methods return iterators over portions of the container (the children of a node of a tree or locators with a given key in a dictionary). JDSL iterators can be traversed only forward; however, they can be reset to start a new traversal.

For simplicity, JDSL iterators have snapshot semantics — they refer to the state of the container at the time the iterator was created, regardless of the possible subsequent modifications of the container. For example, if an iterator is created over all the nodes of a tree and then a subtree is cut off, the iterator still includes the nodes of the removed subtree.

Decorations

Another feature of JDSL is the ability to decorate individual positions of a positional container with attributes (that is, arbitrary objects). This mechanism is more convenient and flexible than either subclassing the position class to add new instance variables or creating global hash tables to store the attributes. Decorations are useful for storing temporary or permanent results of the execution of an algorithm. For example, in a depth-first search traversal of a graph, you can use decorations to (temporarily) mark the vertices being visited and to (permanently) store the computed DFS number of each vertex. We use instances of *java.lang.Object* for both the name and value of each attribute.

Comparators

When using a key-based container, users should be able to specify the comparison relation to be used with the keys. In general, this relation depends on the type of the keys and specific application for which the key-based container is used. Keys of the same type may be compared differently in different applications. One way to meet this requirement is to specify the comparison relation through a comparator object, which is passed to the key-based container constructor and then used by the key-based container every time two keys need to be compared. As Figure 3 shows, JDSL defines three comparator interfaces. A comparator interface is also defined in the Java 2 java.util package, but is not present in JDK 1.1. To maintain the backward compatibility of JDSL with the latter, the JDSL Comparator does not extend the Java Comparator.

Algorithms

JDSL views algorithms as objects that receive the input data as parameters of their execute method, and provide access to the output during or after the execution via additional methods. Most algorithms in JDSL are implemented by applying the template method pattern (see *Design Patterns*, by Erich Gamma et al., Addison-Wesley, 1995). The invariant part of an algorithm is implemented once in an abstract class, deferring the implementation of the steps that can vary to subclasses. These varying steps can be defined either as abstract methods (whose implementation must be provided by a subclass) or as hook methods (whose default implementation may be overridden in a subclass). In other words, algorithms perform generic computations that can be specialized to specific tasks by subclasses.

To illustrate the use of the template method pattern, we examine the JDSL implementation of Dijkstra's single-source shortest path algorithm. The algorithm refers to the edge weights by means of an abstract method that can be specialized depending on how the weights are actually stored or computed in the application at hand.

JDSL Architecture

JDSL currently consists of eight Java packages, each containing a set of interfaces and/or classes. The interfaces for the various containers are organized into two hierarchies — one for the positional containers (Figure 4) and another for key-based containers (Figure 5), with a common root given by interfaces InspectableContainer and Container. Most containers are described by two interfaces — one that contains all the methods to query the container (its name is prefixed with Inspectable), and the other, extending the first, that contains all the methods to modify the container. Inspectable interfaces can be used as variable or parameter types to obtain an immutable view of a container (for instance, to prevent an algorithm from modifying the container it operates on).

• jdsl.core.api. Interfaces and exceptions that compose the API for the core containers (sequences, trees, priority queues, and dictionaries), and for the iterators on their elements, positions, and locators.

- jdsl.core.ref. Implementations of the interfaces in jdsl.core.api. Most implementations have names of the form *{ImplementationStyle}{InterfaceName}*. For instance, *ArraySequence* and *NodeSequence* implement the jdsl.core.api.Sequence interface with a growable array and with a linked structure, respectively. Classes with names of the form *Abstract{InterfaceName}* implement some methods of the interface for the convenience of developers building alternative implementations.
- jdsl.core.algo.sorts. Sorting algorithms that operate on the elements stored in a *jdsl.core.api.Sequence* object. They are parameterized with respect to the comparison rule used to sort the elements, provided as a *jdsl.core.api.Comparator* object.
- jdsl.core.algo.traversals. Traversal algorithms that operate on *jdsl.core.api.InspectableTree* objects. A traversal algorithm performs operations while visiting the nodes of the tree, and can be extended applying the template method pattern.
- jdsl.core.util. This package currently contains a *Converter* class to convert JDSL data structures to Java Collections and vice versa.
- jdsl.graph.api. Interfaces and exceptions that compose the API for the graph container, and for the iterators on its vertices and edges.
- jdsl.graph.ref. Implementations of the interfaces in jdsl.graph.api; in particular, class *IncidenceListGraph* is an implementation of interface jdsl.graph.api.Graph.
- jdsl.graph.algo. Basic graph algorithms, including depth-first search, topological sorting, shortest path, and minimum spanning tree, all of which can be extended applying the template method pattern.

A Sample Application

To illustrate how you can use JDSL, we present a sample application that uses concepts such as the graph and priority queue data structures, locators, decorations, and template method pattern. Specifically, we consider the problem of calculating a minimum-time flight itinerary between two airports. The flight network can be modeled using a directed graph: Each vertex of the graph represents an airport, and each directed edge represents a flight from the origin airport to the destination airport. The problem can be solved by computing a shortest path between two vertices of a directed weighted graph, or determining that a path does not exists. To this purpose, we can suitably modify the classical algorithm by Dijkstra (see "A Note on Two Problems in Connection with Graphs," by E.W. Dijkstra, *Numerische Mathematik*, 1959), which takes as input a graph G with nonnegative edge weights and a distinguished source vertex s, and computes a shortest path from s to any reachable vertex of G. Dijkstra's algorithm maintains a priority queue Q of vertices: At any time, the key of a vertex u in the priority queue is the length of the shortest path from s to u found so far. The priority queue is initialized by inserting vertex s with *key* 0 and all the other vertices with *key*+1 (some very large number). The algorithm repeatedly executes the steps:

1. Remove a minimum-key vertex u from the priority queue and mark it as finished, since a shortest path from s to u has been found.
2. For each edge e connecting vertex u to an unfinished vertex v, if the path formed by extending a shortest path from s to u with edge e is shorter than the shortest known path from s to v, update the key of v (this operation is known as the "relaxation of edge e").

JDSL includes an implementation of Dijkstra's algorithm that applies the template method pattern. The primitive operations of the algorithm are defined by some abstract or overridable methods. The invariant steps of the algorithm are implemented in a few unmodifiable methods that call the primitive operation methods. To specialize the algorithm to the application at hand, you must subclass the algorithm and define or override the primitive operation methods. The abstract class implementing Dijkstra's algorithm is *IntegerDijkstraTemplate* in package jdsl-.graph.algo (see Listings One through Three; for brevity, we removed the Javadoc comments). The simplest way to run the algorithm is by calling *execute(g,source)*, which first initializes the various auxiliary data structures with *init(g,source)* and then repeatedly invokes *doOneIteration( )*. The number of times *doOneIteration( )* is invoked is controlled by *shouldContinue( )*. Instead of calling *execute(g,source)*, another possibility is to call *init(g,source)* directly and then single-step the algorithm by explicitly calling *doOneIteration( )*.

For an efficient implementation of the algorithm, it is important to access a vertex stored in the priority queue in constant time, whenever its key has to be modified. This is possible through the locator accessors provided by the priority queue. In *init(g,source)*, each vertex *u* of the graph is inserted in the priority queue and a locator *uLoc* for the key/vertex pair is returned. Through *setLocator(u,uLoc)*, each vertex *u* is decorated with its locator *uLoc; variableLOCATOR* is used as the attribute name. Later, in *doOneIteration( )*, the locator is retrieved with *getLocator(v)*, in order to access and possibly modify the key of vertex *v;* we recall that the key of *v* is the shortest known distance from source to *v*. In addition to its locator in the priority queue, every unfinished vertex *v* is also decorated with its last relaxed incident edge *uv* through *setEdgeToParent(v,uv)*; variable EDGE_TO_PARENT is used as the attribute name, in this case. When a vertex is finished, this decoration stores the edge to the parent in the shortest path tree, and can be retrieved with *getEdgeToParent(Vertex)*.

Methods *runUntil( )* and *doOneIteration( )* are declared final and thus cannot be overridden. However, they invoke some methods, namely *shouldContinue( )*, *vertexNotReachable(u)*, *shortestPathFound(u,uDist)*, and *edgeRelaxed(u,uDist,uv,uvWeight,v,vDist)*, that may be overridden for special applications. For each vertex *u* of the graph, either *vertexNotReachable(u)* or *shortestPathFound(u,uDist)* is called exactly once, when *u* is removed from the priority queue and marked as finished. In particular, *shortestPathFound(u,uDist)* decorates *u* with *uDist*, the shortest distance from source; *variableDISTANCE* is used as the attribute name. Method *edgeRelaxed(u,uDist,uv,uvWeight,v,vDist)* is called every time an edge *uv* from a finished vertex *u* to an unfinished vertex *v* is examined. The only method whose implementation must be provided by a subclass is abstract method *weight(Edge)*, which returns the weight of an edge. Finally, *distance(Vertex)* lets users query each finished vertex for the shortest distance from source.

JDSL also provides a specialization of Dijkstra's algorithm to the problem of finding a shortest path between two vertices of a graph. This algorithm is implemented by abstract class *IntegerDijkstraPathfinder* (see Listing Four), which extends *IntegerDijkstraTemplate*. The execution of Dijkstra's algorithm is stopped as soon as the destination vertex is finished. To this purpose, *shouldContinue( )* is overridden to return True only if the destination vertex has not been finished yet. Additional methods are provided in *IntegerDijkstraPathfinder* to test, after the execution of the algorithm, whether a path from the source vertex to the destination vertex exists, and in this case, to return it.

Our application for computing a minimum-time flight itinerary between two airports can be implemented as a specialization of *IntegerDijkstraPathfinder*. The distance of each vertex represents, in this case, the time elapsed between the arrival at the corresponding airport and the beginning of the travel. Listing

Five is class *FlightDijkstra*. All it takes to implement our application is to override *incidentEdges( ),* so that only the outgoing edges of a finished vertex are examined, and to define *weight(Edge)*. The weighted graph representing the flight network is a directed graph. Each edge stores, as an element, an instance of *FlightSpecs*, an auxiliary class that provides the departure time and the duration of the corresponding flight. The weight of each edge is not determined before the execution of the algorithm, but depends on the computed shortest distance between the source and origin of the edge. Namely, it is obtained by adding the duration of the flight corresponding to the edge and the connecting time at the origin airport for that flight. (In the sample application we ignore the minimum connecting time requirement, which could be accommodated with minor code modifications.) The algorithm is run by calling *execute(g,source,dest,startTime)*, where *startTime* is the earliest time the passenger can begin traveling. Method *TimeTable.diff(int,int)* simply computes the difference between its two arguments modulo 24 hours.

As you can see, the availability in JDSL of a set of carefully designed and extensible algorithms and data structures makes it possible to implement moderately complex applications with a small amount of code, thus dramatically reducing the development time.

Future Directions

In the current version of JDSL, our emphasis has been on data structures, while only a basic repertory of algorithms has been provided. Future versions will include a wider selection of algorithms such as biconnected components, maximum flow, matching, and graph drawing algorithms. More complex data structures such as topological graphs and planar subdivisions, will be added as well. We also plan to make JDSL data structures thread-safe and serializable. Finally, future versions will include a package for testing whether the (user-provided) implementation of a data structure complies with the interface specification, and a data structure visualization package.

Acknowledgments

This work was supported in part by the U.S. Army Research Office under grant DAAH04-96-1-0013 and the National Science Foundation under grant CCR-9732327. Send correspondence to Luca Vismara, Department of Computer Science, Brown University, 115 Waterman Street, Providence, RI 02912-1910, phone 401-863-7664, fax 401-863-7657, e-mail lv@cs.brown.edu.

DDJ

(Listings begin on page xx.)

Listing One

```
package jdsl.graph.algo;

import jdsl.core.api.*;
import jdsl.core.ref.ArrayHeap;
import jdsl.core.ref.IntegerComparator;
import jdsl.graph.api.*;

public abstract class IntegerDijkstraTemplate {

   // instance variables
   protected PriorityQueue pq_;
   protected InspectableGraph g_;
   protected Vertex source_;
   private final Integer ZERO = new Integer(Ø);
   private final Integer INFINITY = new Integer(Integer.MAX VALUE);
   private final Object LOCATOR = new Object();
   private final Object DISTANCE = new Object();
   private final Object EDGE_TO_PARENT = new Object();

   // abstract instance methods
   protected abstract int weight (Edge e);

   // instance methods that may be overridden for special applications
   protected void shortestPathFound (Vertex v, int vDist) {
      v.set(DISTANCE,new Integer(vDist));
   }
   protected void vertexNotReachable (Vertex v) {
      v.set(DISTANCE,INFINITY);
      setEdgeToParent(v,Edge.NONE);
   }
   protected void edgeRelaxed (Vertex u, int uDist, Edge uv, int uvWeight,
                                                 Vertex v,int vDist) { }
   protected boolean shouldContinue () {
      return true;
   }
   protected boolean isFinished (Vertex v) {
     return v.has(DISTANCE);
   }
   protected void setLocator (Vertex v, Locator vLoc) {
     v.set(LOCATOR,vLoc);
   }
   protected Locator getLocator (Vertex v) {
     return (Locator)v.get(LOCATOR);
   }
   protected void setEdgeToParent (Vertex v, Edge vEdge) {
     v.set(EDGE TO PARENT,vEdge);
   }
```

Listing Two

```
   protected EdgeIterator incidentEdges (Vertex v) {
      return g_.incidentEdges(v,EdgeDirection.OUT ¦ EdgeDirection.UNDIR);
   }
   protected Vertex destination (Vertex origin, Edge e) {
     return g_.opposite(origin,e);
   }
   protected VertexIterator vertices () { return } .vertices();
   }
   protected PriorityQueue newPQ () {
     return new ArrayHeap(new IntegerComparator());
   }
   // output instance methods
   public final boolean isReachable (Vertex v) {
     return v.has(EDGE_TO_PARENT) &&v.get(EDGE_TO_PARENT) !=Edge.NONE;
   }
   public final int distance (Vertex v) throws InvalidQueryException {
     try {
       return ((Integer)v.get(DISTANCE)).intValue();
   }
   catch (InvalidAttributeException iae) {
     throw new InvalidQueryException(v+" has not been reached yet");
   }
}
public Edge getEdgeToParent (Vertex v) throws InvalidQueryException {
   try {
      return (Edge)v.get(EDGE_TO_PARENT);
   }
   catch (InvalidAttributeException iae) {
      String s = (v == source ) ?" is the source vertex" :
                                             " has not been reached yet";
      throw new InvalidQueryException(v+s);
   }
}
// instance methods composing the core of the algorithm
public void init (InspectableGraph g, Vertex source) {
   g_= g;
   source = source;
   pq_= newPQ();
   VertexIterator vi = vertices();
   while (vi.hasNext()) {
     Vertex u = vi.nextVertex();
     Integer uKey = (u == source ) ?ZERO : INFINITY;
     Locator uLoc = pq_.insert(uKey,u);
     setLocator(u,uLoc);
   }
}
```

Listing Three

```
   protected final void runUntil () {
      while (!pq .isEmpty() && shouldContinue())
         doOneIteration();
   }
   public final void doOneIteration () throws InvalidEdgeException {
      Integer minKey = (Integer)pq_.min().key();
      Vertex u = (Vertex)pq_.removeMin();
                   // remove a vertex with minimum distance from the source
      if (minKey == INFINITY)
         vertexNotReachable(u);
      else { // the general case
         int uDist = minKey.intValue();
         shortestPathFound(u,uDist);
         int maxEdgeWeight = INFINITY.intValue() - uDist - 1;
         EdgeIterator ei = incidentEdges(u);
         while (ei.hasNext()) { // examine all the edges incident with u
             Edge uv = ei.nextEdge();
             int uvWeight = weight(uv);
             if (uvWeight < Ø ¦¦ uvWeight > maxEdgeWeight)
                throw new InvalidEdgeException
              ("The weight of "+uv+" is either negative or causing overflow");
             Vertex v = destination(u,uv);
             Locator vLoc = getLocator(v);
             if (pq_.contains(vLoc)) { // v is not finished yet
                int vDist = ((Integer)vLoc.key()).intValue();
                int vDistViaUV = uDist+uvWeight;
                if (vDistViaUV < vDist) { // relax
                   pq_.replaceKey(vLoc,new Integer(vDistViaUV));
                   setEdgeToParent(v,uv);
                }
                edgeRelaxed(u,uDist,uv,uvWeight,v,vDist);
             }
         }
     }
}
public final void execute (InspectableGraph g, Vertex source) {
   init(g,source);
   runUntil();
}
public void cleanup () {
   VertexIterator vi = vertices();
      while (vi.hasNext()) {
         vi.nextVertex().destroy(LOCATOR);
         try {
           vi.vertex().destroy(EDGE_TO_PARENT);
           vi.vertex().destroy(DISTANCE);
         }
         catch (InvalidAttributeException iae) { }
      }
   }
} // class IntegerDijkstraTemplate
```

Listing Four

```
package jdsl.graph.algo;

import jdsl.core.api.*;
import jdsl.core.ref.NodeSequence;
import jdsl.graph.api.*;
import jdsl.graph.ref.EdgeIteratorAdapter;

public abstract class IntegerDijkstraPathfinder
                               extends IntegerDijkstraTemplate {
// instance variables
private Vertex dest_;

// overridden instance methods from IntegerDijkstraTemplate
protected boolean shouldContinue () {
    return !isFinished(dest_);
}
// output instance methods
public boolean pathExists () {
   return isFinished(dest_);
}
public EdgeIterator reportPath () throws InvalidQueryException {
   if (!pathExists())
   throw new InvalidQueryException("No path exists
                                         between "+source +" and "+dest_);
   else { Sequence retval = new NodeSequence();
      Vertex currVertex = dest_;
      while (currVertex != source_) {
         Edge currEdge = getEdgeToParent(currVertex);
         retval.insertFirst(currEdge);
         currVertex = g_.opposite(currVertex,currEdge);
      }
      return new EdgeIteratorAdapter(retval.elements());
   }
}
// instance methods
public final void execute (InspectableGraph g, Vertex source, Vertex dest) {
   dest_= dest;
   init(g,source);
   if (source != dest )
      runUntil();
   }
} // class IntegerDijkstraPath_nder
```

Listing Five

```
import jdsl.graph.api.*;
import jdsl.graph.algo.IntegerDijkstraPath_nder;
import support.*;

public class FlightDijkstra extends IntegerDijkstraPathfinder {
// instance variables
private int startTime ;
// overridden instance methods from IntegerDijkstraPathfinder
protected int weight (Edge e) {
   FlightSpecs eFS = (FlightSpecs)e.element();
                        // the flightspecs for the flight along edge e
   int connectingTime =
       TimeTable.diff(eFS.departureTime(),startTime_+distance(g_.origin(e)));
   return connectingTime+eFS.flightDuration();
}
protected EdgeIterator incidentEdges (Vertex v) {
   return g_.incidentEdges(v,EdgeDirection.OUT);
}
// instance methods
public void execute (InspectableGraph g, Vertex source,
                                              Vertex dest, int startTime) {
   startTime = startTime;
   super.execute(g,source,dest);
   }
}
```

DDJ

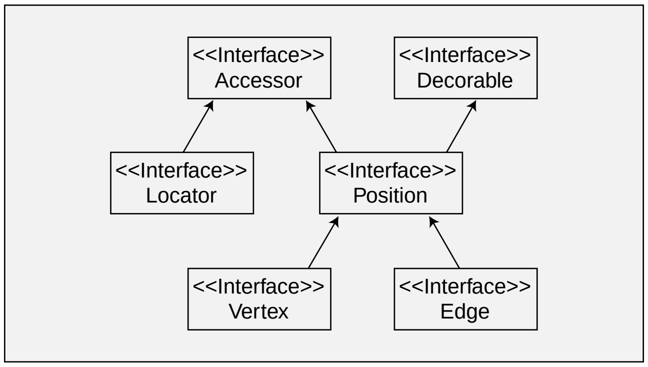

Figure 1: *The accessors interface hierarchy.*

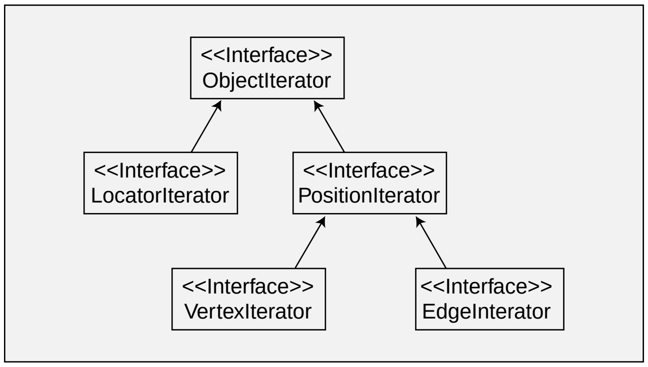
<<Interface>> ObjectIterator
<<Interface>> LocatorIterator
<<Interface>> PositionIterator
<<Interface>> VertexIterator
<<Interface>> EdgeInterator

Figure 2: *The iterators interface hierarchy.*

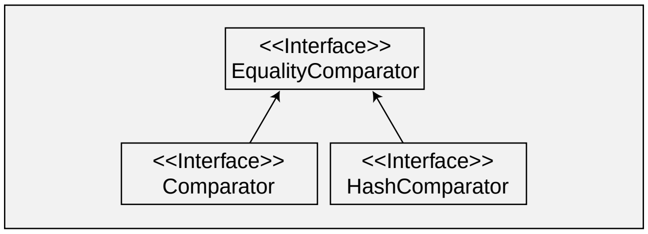
<<Interface>> EqualityComparator
<<Interface>> Comparator
<<Interface>> HashComparator

Figure 3: *The comparators interface hierarchy.*

	JC	JGL	GFC	JDSL
Sequences (lists, vectors)	x	x	x	x
Trees			x	x
Priority queues (heaps)		x		x
Dictionaries (hash tables, red-black trees)	x	x		x
Sets			x	
Graphs			x	x
Templated algorithms				x
Sorting algorithms	x	x		x
Data permutation algorithms		x		
Graph traversals			x	x
Topological sorting				x
Shortest path, Minimum spanning tree				x
Graph drawing algorithms			x	
Accessors (positions and locators)				x
Iterators	x	x		x
Range views	x	x		
Decorations (attributes)			x	x
Thread-safety and serializability	x	x		

Table 1: *Comparing the Java Collections (JC), Generic Library for Java (JGL), Graph Foundation Classes for Java (GFC), and Data Structures Library in Java (JDSL).*

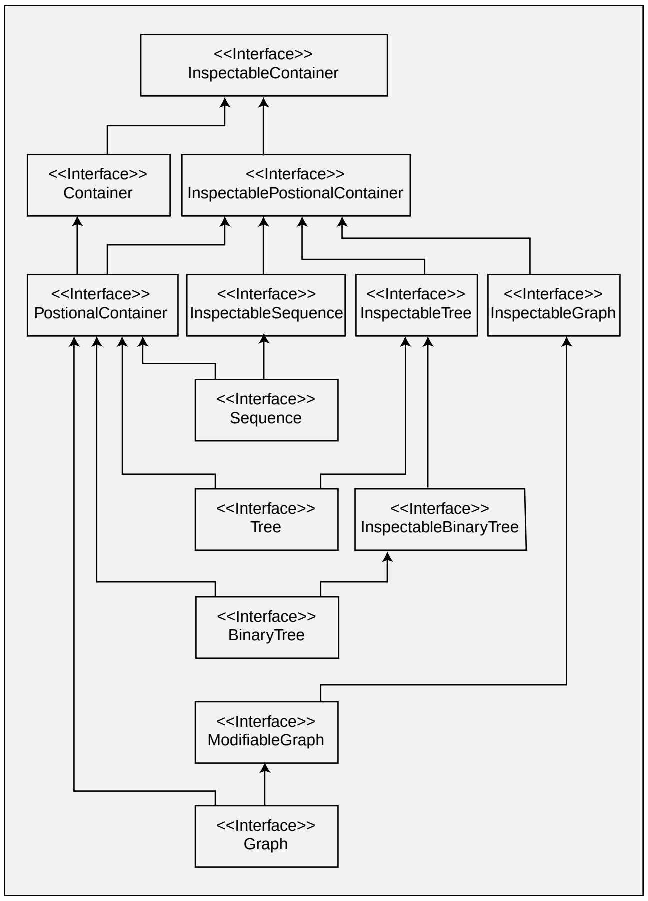
<<Interface>> InspectableContainer
<<Interface>> Container
<<Interface>> InspectablePostionalContainer
<<Interface>> PostionalContainer
<<Interface>> InspectableSequence
<<Interface>> InspectableTree
<<Interface>> InspectableGraph
<<Interface>> Sequence
<<Interface>> Tree
<<Interface>> InspectableBinaryTree
<<Interface>> BinaryTree
<<Interface>> ModifiableGraph
<<Interface>> Graph

Figure 4: *The positional containers interface hierarchy.*

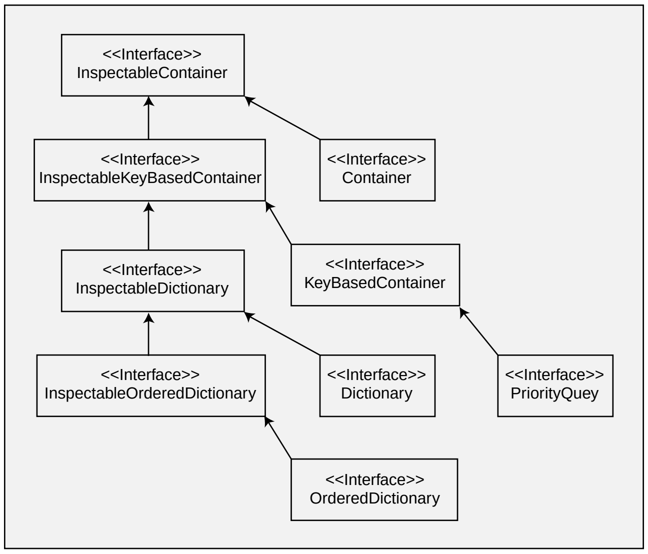
<<Interface>> InspectableContainer
<<Interface>> InspectableKeyBasedContainer
<<Interface>> Container
<<Interface>> KeyBasedContainer
<<Interface>> InspectableDictionary
<<Interface>> InspectableOrderedDictionary
<<Interface>> Dictionary
<<Interface>> PriorityQuey
<<Interface>> OrderedDictionary

Figure 5: *The key-based containers interface hierarchy.*