# Accessing the Internal Organization of Data Structures in the JDSL Library


| Michael T. Goodrich | Mark Handy | Benoît Hudson | Roberto Tamassia |
|---|---|---|---|
| Dept. of Comp. Sci. | Dept. of Comp. Sci. | Dept. of Comp. Sci. | Dept. of Comp. Sci. |
| Johns Hopkins Univ. | Brown Univ. | Brown Univ. | Brown Univ. |
| Baltimore, MD 21218 | Providence, RI 02912 | Providence, RI, 02912 | Providence, RI 02912 |
| goodrich@cs.jhu.edu | mdh@cs.brown.edu | bh@cs.brown.edu | rt@cs.brown.edu |



**Abstract**

Libraries of data structures allow developers to concentrate on the new aspects of their project, instead of having to reinvent solutions to previously solved problems. Many applications require data structures that allow efficient access to their internal organization and to their elements. This feature has been implemented in some libraries with *iterators* or *items*. We present an alternative implementation, used in the Library of Data Structures for Java (JDSL). We refine the notion of an item and split it into two related concepts: *positions* and *locators*. Positions are an abstraction of a pointer to a node or an index into an array; they provide direct access to the in-memory structure of the container. Locators add a level of indirection and allow the user to find a specific element even if the position holding the element changes.


## 1 Introduction

In using a data structure, the user must be granted some form of access to its elements, preferably an efficient way. In some cases, the access can be very limited: in a stack, for example, where we can only look at the top element. But in many cases, we need a more general mechanism to allow the user some handle on the elements. In an array structure, we can use indices. In a linked structure, we can use pointers to the nodes.

From the perspective of object-oriented design, however, we need to restrict access to the internals of the data structures. Otherwise, the user can make a container invalid by modifying data required to maintain the consistency of the data structure. For instance, if a list returns to the user pointers to its nodes, the user should not be able to modify the successor and predecessor pointers directly.

### 1.1 Previous Work

Two current methods of safely granting access to the internals of data structures are by the use of *iterators* and by returning *items*. STL [18], JGL [14] and the java.util.Collection hierarchy of Java 1.2 [12] use iterators. LEDA [15, 16, 17] uses items.

Iterators provide the means to iterate over a collection of elements in some linear order. At any time, an iterator is essentially a pointer to one of the elements; this pointer can be modified by using the incrementing functions the iterator provides. In C/C++, a pointer (other than to void) fits this profile: it points to a position (an address in memory), and the increment operation '++' makes the pointer point to the next element in an array. In a linked list, a simple class with a pointer to a node of the list will do. The access function will return the element of the node, and the increment function will move to the next node in the list. This discussion is of an STL *input iterator*. Other iterators in the STL hierarchy can also move backwards, write to the current element, and so on. The capabilities of iterators do fulfill the requirement to hide the data structure organization from the user, while still allowing the user to refer to an element efficiently. They also make it easy to act on every element of a list.

Iterators, however, have their limits. For example, if a program has an iterator over a container, and subsequently modifies the container, the question arises as to how the iterator will behave. The modification may be reflected by the iterator, or it may not, or the iterator may simply be invalidated. Furthermore, iterators work well on linearly arranged data, but it is not clear how to extend them to non-linear data structures, such as trees and graphs.

Items, as they are referred to in LEDA, are the nodes of a data structure. In order to keep the user from having access to the internal structure, all fields of an item are private; all access is done through the container, passing in the item. This works quite well to describe arbitrary linked structures, including linear data structures. This approach does not preclude the use of iterators: one can easily be written by having it store a current item, which it changes on incrementing. For an array-based structure, items work less well. In LEDA, there are none—one uses indices instead. Another possibility would be to wrap a very simple object around an index. While this may use more memory, the user sees a consistent application of the concept.

Items are not as appropriate in some data structures, however—namely, those in which the container maintains an ordering (rather than requiring the user to define any ordering). For example, consider a binary heap [4]. Inserting an element into a heap may cause an up-heap operation to occur. In this operation, the nodes do not move, but the elements stored at the nodes do (the inserted element is swapped up the tree). A user who had a handle on an item may find the item's element unexpectedly changed as a result of this operation.

## 1.2 Positions and Locations

We have developed a pair of notions related to that of items. On the one hand, there is a concept of a *position* within a data structure; on the other hand, there is another concept of *location* of an element within a data structure. A position is a topological construct: one position is defined as being before another, or the left child of another, and so on. The user is the one who decides the relationships between positions; a data structure cannot change its topology unless the user requests a change specifically. An element always has some position at which it can be found, but as the element moves about in its container, or even from container to container, its position changes. We introduce the concept of a location, which remains bound to the element as it changes position. Whereas the position-to-element mapping can change without the user's knowledge, the location-to-element mapping cannot. Thus positions most closely resemble items, and the distinction between position and location is new.

We illustrate the distinction between positions and locations using the distinction between a binary tree and a binary heap. Drawings of these two data structures might look the same, but the semantic differences are important: An unrestricted binary tree allows the user to modify the connections between nodes, and to move elements from node to node arbitrarily. We call a data structure of this type *positional*; its interface is written in terms of positions. A heap, on the other hand, manages the structure of the tree and the placement of the elements on behalf of the user, based on the key associated with each element. It prevents the user from arbitrarily adjusting the tree; instead, it presents to the user a restricted interface appropriate to a priority queue. We call a data structure of this type *key-based*; its interface is written in terms of locators. A locator guarantees access to a specific element no matter how the heap modifies either the structure of the tree or the position at which the element is stored.

In both cases, the container provides access to its internal organization. A position provides direct access to the structure of the container. A locator provides access only to the order (or other properties) maintained by the container.

A binary tree would support operations like these:

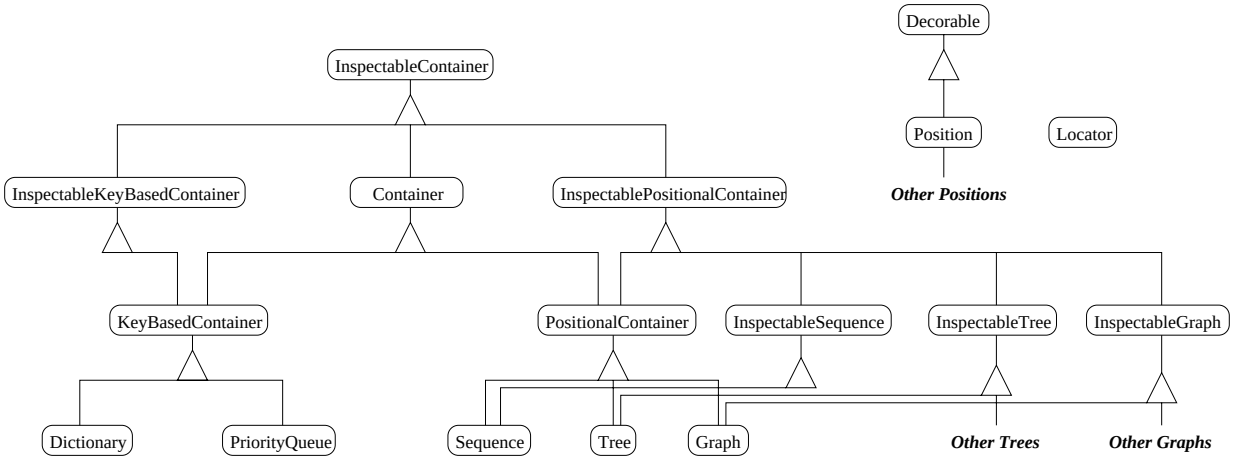
```
leftChild (Position internalNode) returns Position;
cut (Position subtreeRoot) returns BinaryTree; // removes and returns a subtree
swap (Position a, Position b); // exchanges elements and locators
```

A heap would support operations like these:

```
min() returns Locator; // top of the heap
insert (Object key, Object element) returns Locator; // makes new locator
replaceKey (Locator loc, Object newKey) returns Object; // old key
```

## 2 Position and Locator interfaces in JDSL

JDSL, the Data Structures Library for Java, is a new library being developed at Brown and at Johns Hopkins, which seeks to provide a more complete set of data structures and algorithms than has previously been implemented in Java. Our design philosophy is closer to that of LEDA and CGAL [6] than to that of STL and JGL. In addition to the standard set of vectors, linked lists, priority queues, and dictionaries, we provide trees, graphs, and others. We also have algorithms to run on many of the data structures, and are developing a set of fundamental data structures for geometric computing. A partial interface hierarchy follows.




As we see from the hierarchy, JDSL is split into two major parts: KeyBasedContainers such as dictionaries and priority queues, and PositionalContainers such as sequences and graphs. In KeyBasedContainers, elements are arranged by the container based on a key associated with each element. Each *(key,element)* pair is stored at a position within the container; but the position may change without warning to the user. Thus elements are accessed via their location, embodied in the Locator interface. In PositionalContainers, positions are held in a topological relationship to each other. We refer to these using the Position interface.

### 2.1 Positions

Positions are the basic block of all PositionalContainers. They are most commonly used to represent nodes in a linked structure (such as a linked list), or indices in a matrix structure (such as a vector). In data structures which have different types of positions—such as vertices and edges in graphs—we use empty sub-interfaces of Position useful only for type-checking purposes. While positions are closely tied to the internal workings of their container, their public interface is limited, so that they are safe to return to user code. Most operations must actually be done by calling upon the Position's container, passing in the position as an argument—for example, Sequence.after(Position), which returns the position following the argument in the list. Only operations applicable to positions from any container are included in the interface:

```
public interface Position extends Decorable {
  public Object element() throws InvalidPositionException; // return the stored element
  public Locator locator() throws InvalidPositionException; // return the locator for the element
  public Container container() throws InvalidPositionException; // return the container of this position
  public boolean isValid(); // return true if the position is valid; i.e. is still in its container
}
```

In most implementations, the position classes are inner classes of the container that will use them, and have private methods allowing access by the container to their internals. Positions are thus most closely related to LEDA's items, which have only private fields but are friends of their containers—the main difference is that LEDA requires absolutely all accesses to the position to be through the container. Unlike an iterator,

a position is always tied to a particular node, and cannot be used directly to traverse a data structure. Instead, an iterator would use a pointer to a position as its method of indexing into the data structure.

Since positions are so strongly tied to the internal structure of the data structures which contain them, they have no meaning once removed from a container. Hence, a deleted position is marked as invalid and any subsequent use of it will raise an exception—excepting only Position.isValid(), which is intended for the purpose of having a clean way of checking validity.

As an example of using Positions in a simple application, we can look at code which reverses a Sequence (such as a linked list or a vector):

```
void reverse(Sequence S) {
    if(S.isEmpty()) return;

    Position front = S.first(), back = S.last();
    for(int i=0; i<S.size()/2; i++) {
        S.swap(front, back);
        front = S.after(front);
        back = S.before(back);
    }
}
```

## 2.2 Locator

From the point of view of the user, Locators are essentially pointers to elements. They allow the user to efficiently locate an element within a data structure, typically in constant-time, and to make a request of a data structure to act on an element. A locator provides a contract to its user that it will always be associated with a specific element even if the position of that element changes.

```
interface Locator extends Decorable {
  public Object element(); // return the element associated with this locator
  public Container container(); // return the container which currently holds this locator
  public boolean isContained(); // return whether the locator currently has a container
  public Object invalidate(); // permanently make all accesses to this locator invalid
  public boolean isValid(); // return true unless invalidate() has been called
}
```

In order to be useful to a container, a locator must have a method of keeping track of some kind of position within the container object. Otherwise, a container, given a locator, could not efficiently determine where the element is within the data structure. We find that in fact, all containers are at some level positional: they all arrange data in memory, and memory itself is positional. For instance, a dictionary might use a tree, while a priority queue might use a vector.

Locators are often used to access *(key, element)* pairs within KeyBasedContainers. Upon inserting an element, a locator is created and returned to the user. The locator can also be retrieved, in a dictionary, by calling find(Object). Once it has been retrieved, the user can get the element they were searching for, or they can use the locator to efficiently refer to the pair—for example, in replaceKey(Locator, Object), which changes the key of a given locator and adjusts the container if needed.

JDSL implements the concept of *universality* of locators across containers. That is, a single implementation of Locator exists, and all our containers use instances of this single class as their locators. This class, which we call UniversalLocator, stores a Position within the container which currently contains it (the position is undefined when the locator is uncontained). Public methods of the class allow access to the state of the locator. Barring downcasts, the user is prevented from using these because the return type of methods is Locator.

Universality allows a user to move locators from one data structure to another, as long as all the data structures support UniversalLocator. This is useful, for example, in sorting applications, where the position-to-element binding is broken, but the locator-to-element binding need not be. By inserting and removing locators rather than elements, the author of the sort can allow the user to keep useful handles to the elements through the locators. For example, consider the following code for a PQ-Sort:

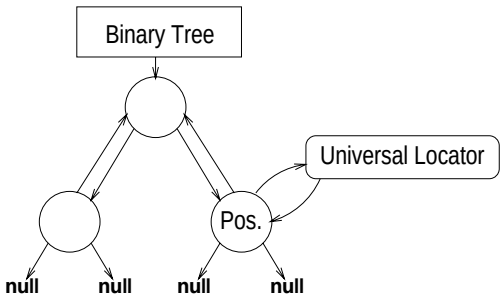
```
public void sort(Sequence S, Comparator c) {
    PriorityQueue Q = new VectorHeap(c); // use your favorite PriorityQueue

    // remove all elements from S, but retain the same locators
    while(!S.isEmpty()) {
        Locator loc = S.first().locator() ; // loc is in S
        S.removeFirst(); // after this, loc is in no container
        Q.insert(loc); // now loc is in Q
    }

    // remove all elements from Q in ascending order
    while(!Q.isEmpty()) {
        S.insertLast(Q.removeMin()); // move the locator from Q to S
    }
}
```

Universal locators require some overhead: a Position object needs to be created. In most cases, this is acceptable: preliminary experiments indicate that our red-black tree, written under the paradigm of universal locators, is at least as fast as the red-black tree in JGL or JDK 1.2 [3]. But in order to allow potentially faster or smaller containers, universality is not required. So instead of storing a Position, the locator for an optimized data structure could itself be the position—a red-black tree locator could store its color, children, and parent.

The following schematic figure shows an implementation of a binary tree, using universal locators.




## 2.3 Decorable

Many algorithms need to store extra state associated with the positions or elements of the data structures they use. For example, in a breadth-first or depth-first search over a graph, we need to mark nodes as visited. Essentially, we want to add decorations or attributes to the positions of our data structures.

We can imagine the system as a two-dimensional matrix of values, with positions indexing the rows and attribute names indexing the columns [20]. We need to be able to find the value of a specific attribute for a specific position. The three solutions commonly used now are:

1. to copy the data structure into one which has the extra instance variables,
2. to use an external dictionary indexed by the positions (the *column-based* option), and
3. to associate an internal dictionaries with each positions (the *row-based* option); the dictionaries are indexed by attribute names.

The first solution clearly involves a large amount of copying of data structures. In a situation where a large number of relatively quick algorithms are being used, this could affect speed significantly.

The second solution is implemented in LEDA's node_map, edge_map, node_array, and edge_array. In JDSL, the column-based solution can be implemented simply by instantiating a hash table or other dictionary, using the hashcode() function of the positions as the key into the dictionary. Thus, no library support is required for the column-based solution.

JDSL supports the third solution; library support is required if the row-based solution is to exist at all. Position extends the Decorable interface, which requires that all positions have a dictionary associated with them. This is also the approach taken in the ffGraph library [7], which calls the attributes "labels". Under the assumption that there are more positions than there are decorations, the row-based solution is asymptotically faster than the column-based in the worst case. In the average case, the efficiencies are the same.

### 2.4 Enumerations

While one use of iterators is to encapsulate a position, they are also useful in iterating over an entire container. To support this functionality, JDSL requires that its data structures provide Enumerations over interesting sets of elements, positions, and locators.

One difficulty we noted with an iterator is its behavior upon modification of the underlying container. To avoid the issue, we specify that Enumerations provide a *snapshot* of the data. That is, subsequent modifications to the data structure over which we are enumerating do not modify outstanding enumerations. In general, a user who asks for an enumeration will use every element of it, so we are not affecting asymptotic efficiency. Also, we have developed optimizations to further reduce the cost (see section 4).

## 3 Example: Dijkstra's Algorithm

We present, as an example, a simple implementation of Dijkstra's algorithm for finding shortest paths; the implementation is based on [4]. The algorithm is typically used to illustrate the programming methodology of a library [20]. We assume the edges of the graph are weighted, using the Decorable mechanism, with non-negative Integer weights.

```
package examples;

import jdsl.core.api.*;
import jdsl.graph.api.*;
import jdsl.core.ref.*;
import java.util.Enumeration;

/**
 * Dijkstra's algorithm, which runs over an InspectableGraph, and uses integer weights.
 * The output is a list of edges which define the shortest path from s to t.
 */
public class Dijkstra {

    // data structures we will be using
    private InspectableGraph graph_;
    private PriorityQueue pq_ = new VectorHeap(new IntegerComparator());
    private Sequence output_ = new NodeSequence();

    // the source and destination of the path
    private Vertex s_, t_;

    // keys for decorations
    private Object weight_ ;
    private Object locator_ = new Object();
    private Object incoming_ = new Object();

    // "infinity" and zero for the purposes of this program
    private static final Integer INFINITE = new Integer(Integer.MAX_VALUE);
    private static final Integer ZERO = new Integer(0);

    // Initialize the object, and run the algorithm.
    public Dijkstra(InspectableGraph g, Vertex s, Vertex t, Object weight) {
        graph_  = g;
        weight_ = weight;
        s_ = s; t_ = t;
```

```
    initialize();
    run();
    buildPath();
    cleanup();
}

// Return the path as an enumeration of edges from s to t.
public Enumeration getPath() { return output_.elements(); }

// Create appropriate decorations and put vertices into the heap
private void initialize() {
    for(Enumeration verts = graph_.vertices(); verts.hasMoreElements(); ) {
        Vertex v = (Vertex)verts.nextElement();
        v.create(locator_ , pq_.insert(INFINITE, v));
        v.create(incoming_, null);
    }

    // we have set s to have infinite distance, but it has 0 distance
    pq_.replaceKey(locator(s_), ZERO);
}

// Clean up: destroy all decorations we created.
private void cleanup() {
    for(Enumeration verts = graph_.vertices(); verts.hasMoreElements(); ) {
        Vertex v = (Vertex)verts.nextElement();
        v.destroy(locator_);
        v.destroy(incoming_);
    }
}

// Run the core of the algorithm
private void run() {
    while( !pq_.isEmpty() ) {
        Vertex v = (Vertex) pq_.removeMin();

        if(v==t_) return; // if true, we've found the shortest path to t

        int distance = distance(v);

        // for all outgoing edges...
        for(Enumeration edges = graph_.outIncidentEdges(v);
                edges.hasMoreElements() ; ) {
            Edge e = (Edge) edges.nextElement();
            Vertex dest = graph_.destination(e);

            int cumulative = weight(e) + distance;

            if( cumulative < distance(dest) ) { // relax the edge if applicable
                Integer newdist = new Integer(cumulative);
                pq_.replaceKey( locator(dest), newdist);
                dest.set(incoming_, e);
            }
        }
    }
}

// Build the list of edges from source to destination
private void buildPath() {
    Vertex v = t_; // we're going backwards...
    while(v!=s_) {
        Edge e = incoming(v);
        output_.insertFirst(e); // so we insert at the head
        v = graph_.origin(e);
    }
}

// some accessors
private Locator locator(Vertex v) { return (Locator)v.get(locator_); }
```

```
    private int distance(Vertex v) { return ((Integer)pq_.key(locator(v))).intValue(); }
    private Edge incoming(Vertex v) { return (Edge)v.get(incoming_); }
    private int weight(Edge e) { return ((Integer)e.get(weight_)).intValue(); }

}
```

Here we use both positions and locators. The positions of a graph are its edges and vertices—this is reflected in JDSL by having Edge and Vertex extend Position. It is by using these that the algorithm traverses the graph. The locators we use are to be able to find the vertices in the priority queue. We insert into the queue pairs *(distance, vertex)*, which represent the shortest yet known distance to the vertex. A vertex is only chosen if it is the closest vertex to the source that we have not yet encountered. When we choose a vertex, we discover all its edges. Through these edges, we may find a shorter path to another vertex. If we do, we need to update the key of the vertex within the queue (since the vertex should be removed from the queue earlier than previously believed), which we can do efficiently because we kept a locator to the vertex.

# 4 Efficiency

JDSL's use of locators and positions is asymptotically efficient; algorithms do not suffer increased complexity, either in the worst case or in the average case, because access via locators and positions requires constant time. But a library which claims to be all-powerful invariably suffers from constant-factor efficiency losses: there are clear tradeoffs among power, elegance, and speed. We have developed some optimizations which should make JDSL competitive with other data structures libraries in terms of speed. Experiments are in progress to compare actual performance of JDSL with that of other libraries.

One concern is caused by the requirement for each element to be associated with a locator. This obviously creates a large number of extra objects. Further, in PositionalContainers, the locators often go unused—they may be powerful, but many applications do not need the power. Therefore, most data structures in the library follow an *allocate-on-use* policy for creating locators. If the user asks to get a locator, via the Position.locator() method, for example, one is created. Otherwise, we short-circuit and store an element directly. This can save considerable time and space, especially in situations where we are creating and destroying large numbers of positions, without ever querying them for a locator. The only cost here is a check and conditional branch whenever the element or locator is queried from a position, to see if the locator has been allocated or not.

Another concern is the time used in building enumerations. Since these are snapshots of the data structures, creating them will take time at least linear in the number of elements to be included in the enumeration. Here we have applied a *copy-on-write* policy for some data structures: when an enumeration is asked for, we create an array which stores all the objects over which to enumerate, then return an enumeration over that array. If the same enumeration is asked for again, we can simply return another enumeration object, over the same array—in constant time. If a call is made to a function which modifies the correct data to return, we discard our array (Java's garbage-collection scheme takes care of disposing of it). In this way, we are optimizing for having a number of read-only calls in between phases of modification. This is in fact realistic: a common use of a data structure is to fill it with data, then run some algorithms on it which only inspect the data. The cost of this method is none in terms of time, but in most implementations, we have to keep an extra copy of the data around between the time an enumeration has been called for and the next modification. If we have multiple enumerations simultaneously, however, we save space since we share a single copy among them. Furthermore, in array-based data structures, the array can be used as the single copy, so that even the first enumeration call can be done in constant-time; only if the container is modified while there are enumerations outstanding do we actually need to copy any data.

# 5 Experience with JDSL

JDSL has a companion *teach* version [13], more suited for pedagogical use than the research library is. It has been used successfully in first-year, one-semester, CS2-level classes at Brown [5] and Johns Hopkins, and is the library discussed in [10]. At Brown, over a hundred CS2 students implemented the following data structures and algorithms using the JDSL *teach* library:

- Sequence, implemented with a circular array
- Binary search and quicksort of a Sequence
- Binary tree
- Binary heap, reusing the binary tree
- Red-black tree, reusing the binary tree
- Hash table
- Rabin-Karp string-searching algorithm
- Convex hull algorithm, using package wrap
- Spanning tree of an undirected graph
- Directed graph
- Prim's minimum-spanning-tree algorithm
- Dijkstra's shortest-path algorithm

The methodology of the library allows projects that are more theoretically sophisticated than were possible in past years, and more full-featured. Data structures met the interfaces specified by JDSL. For instance, most data structures included a full suite of modifiers (insertion, deletion, and replacement) and thorough error handling via exceptions. Students were supported by visualizers and testers written within JDSL [1].

A number of implementation projects have been written based on the research version of the library, as well. Various point location algorithms have been implemented [19]. A planar map (an embedded planar graph, or EPG) has been implemented [11], with operations that preserve planarity. The EPG is built on top of an ordered graph from the library. On top of the EPG, there are algorithms for orthogonal drawings of graphs [9] and for finding the shortest path between two points in the interior of an arbitrary polygon [2].

## Acknowledgments

We would like to thank the entire JDSL team for their work on the project—in particular, Andy Schwerin and Maurizio Pizzonia for their constructive criticism of this paper, and John Kloss for helpful comments regarding topics related to the paper.

Work supported in part by the U.S. Army Research Office under grant DAAH04–96–1–0013 and by the National Science Foundation under grants CCR–9625289, CCR–9732327, and CDA-9703080.

# References


[1] R. Baker, M. Boilen, M. T. Goodrich, R. Tamassia, and B. A. Stibel. Testers and visualizers for teaching data structures. Manuscript.

[2] J. Beall. Shortest path between two points in a polygon. http://www.cs.brown.edu/courses/cs252/projects/jeb/html/cs252proj.html.

[3] M. Boilen, A. Schwerin, and J. Kloss. Personal communication.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[5] CS 16 home page. http://www.cs.brown.edu/courses/cs016.

[6] A. Fabri et al. The cgal kernel: A basis for geometric computation. In *Proc. 1st ACM Workshop on Appl. Comput. Geom.*, pages 97–103, May 1996.

[7] ffGraph home page. http://www.fmi.uni-passau.de/~friedric/ffgraph/main.shtml.

[8] N. Gelfand, M. T. Goodrich, and R. Tamassia. Teaching data structure design patterns. In *Proc. SIGCSE*, 1997.

[9] N. Gelfand and R. Tamassia. Algorithmic patterns for graph drawing. In *Proc. Graph Drawing '98*. Springer-Verlag, to appear.

[10] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. Wiley, New York, NY, 1998.

[11] D. Jackson. The TripartiteEmbeddedPlanarGraph. Manuscript.

[12] Java 1.2beta API. http://java.sun.com/products/jdk/1.2/docs/api/index.html.

[13] JDSL home page. http://www.cs.brown.edu/cgc/jdsl.

[14] JGL - the generic collection library for java. http://www.objectspace.com/jgl.

[15] K. Mehlhorn and S. Näher. Algorithm design and software libraries: Recent developments in the LEDA project. In *Algorithms, Software, Architectures, Information Processing 92*, volume 1, pages 493–505, Amsterdam, 1992. Elsevier Science Publishers B.V. North-Holland.

[16] K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *Commun. ACM*, 38(1):96–102, 1995.

[17] S. Näher and C. Uhrig. *The LEDA Manual User Manual*, 1995.

[18] B. Stroustrup. *The C++ Programming Language (3rd Edition)*. Addison-Welsey, Reading, MA, 1997.

[19] R. Tamassia, L. Vismara, and J. E. Baker. A case study in algorithm engineering for geometric computing. In *Proc. Workshop on Algorithm Engineering*, pages 136–145, 1997.

[20] K. Weihe. Reuse of algorithms: Still a challenge to object-oriented programming. In *Proc. OOPSLA '97*, pages 34–48, 1997.