An Overview of JDSL 2.0, the Data Structures Library in Java*

Roberto Tamassia[†‡] *Michael T. Goodrich*[§‡] *Luca Vismara*[†] *Mark Handy*[†]
Robert Cohen[†‡] *Benoît Hudson*[¶] *Ryan S. Baker*[†] *Natasha Gelfand*[‖]
Galina Shubina[†] *Michael Boilen*[†] *Maurizio Pizzonia*[**] *Keith Schmidt*[†]
Ulrik Brandes[††] *Lucy Perry*[†] *Andrew Schwerin*[†] *David Ellis*[†]

http://www.cs.brown.edu/cgc/jdsl/
jdsl@cs.brown.edu

(August 31, 2005)

Abstract

We introduce version 2.0 of the *Data Structures Library in Java* (*JDSL*). In addition to basic data structures, such as lists and dictionaries, JDSL includes implementations of a variety of complex data structures, such as trees, graphs, and priority queues, with powerful and flexible access to the elements stored, both by means of traditional iterators and by means of new types of accessors called *positions* and *locators*.

*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 and CCR–9732327.

[†]Center for Geometric Computing, Department of Computer Science, Brown University, 115 Waterman Street, Providence, RI 02912–1910, USA.

[‡]Algomagic Technologies, Inc., 31 Audrey Road, Belmont, MA 02478, USA.

[§]Center for Geometric Computing, Department of Computer Science, The Johns Hopkins University, 3400 N. Charles Street, Baltimore, MD 21218, USA. Work on this project performed by this author as a consultant to Algomagic Technologies, Inc., and as a coauthor of the book *Data Structures and Algorithms in Java*.

[¶]NASA Ames Research Center, Moffett Field, CA 94035-1000, USA. Work performed while this author was with the Center for Geometric Computing, Department of Computer Science, Brown University.

[‖]Computer Science Department, Stanford University, Stanford, CA 94305-9025, USA. Work performed while this author was with the Center for Geometric Computing, Department of Computer Science, Brown University.

[**]Dipartimento di Informatica e Automazione, Università degli Studi di Roma Tre, Via della Vasca Navale 79, 00146 Roma, Italy. Work performed while this author was with the Center for Geometric Computing, Department of Computer Science, Brown University.

[††]Faculty of Mathematics and Computer Science, University of Konstanz, Fach D 188, D-78457 Konstanz, Germany. Work performed while this author was with the Center for Geometric Computing, Department of Computer Science, Brown University.

1 Introduction

Computer programs organize information into *data structures* and process it according to *algorithms*. Thus, providing component libraries of efficient data structures and algorithms can enable rapid software development for advanced applications.

1.1 Benefits of Libraries

The benefits of using components for advanced algorithms and data structures are significant, resulting in software that is much more reliable and faster to code than alternate choices. Indeed, reinventing such components presents major risks to developers:

- *Unreliability*: complex algorithms and data structures are difficult to implement and even more difficult to prove correct, since standard software testing methods are often inadequate for them.
- *Insufficient knowledge*: programmers are typically exposed to only the most elementary algorithms and data structures as computer science students. Thus, little advanced and newly discovered algorithmic knowledge reaches the general software development community.
- *Long development time*: advanced algorithms and data structures usually require more time to implement than simple, slow-running solutions.

While the final version of a software application may use very specialized data structures, crafted exactly to the purpose at hand, the availability of a component library substantially speeds up software development.

1.2 Data Structure Libraries in C++

A major development in software engineering was the introduction of the *Standard Template Library* for C++ (STL) [13], which is the first widely used library of data structures and is now part of the C++ standard.

While STL includes only elementary data structures, such as lists and dictionaries, more advanced libraries are also available to C++ programmers. Among them: the Library of Efficient Data Structures and Algorithms (LEDA) [11, 12], which includes more complex data structures, such as trees and graphs, and a variety of sophisticated algorithms; the Generic Graph Component Library (GGCL) [16], which applies the principles of generic programming from STL to a library of graph data structures and algorithms.

Finally, other C++ libraries exist for specialized applications, such as the Computational Geometry Algorithm Library (CGAL) [5] for geometric computing, and the Graph Drawing Toolkit [4] and the library of Algorithms for Graph Drawing (AGD) [14] for graph drawing applications.

1.3 Data Structure Libraries in Java

Java is evolving into a premier development language for advanced software applications, particularly for the Internet. A small library of data structures and algorithms, which we refer to as *Java Collections* (JC) is included in the standard Java package `java.util` [17]. An alternative library is the *Generic Library for Java* (*JGL*) by ObjectSpace [15], which is patterned after STL. Both the Java Collections and JGL provide implementations of basic data structures such as maps, sets, dictionaries, and sequences. JGL also provides a considerable number of template-based algorithms for permuting data. The *Graph Foundation Classes for Java* (*GFC*) by alphaWorks [1] is a framework for programming with graphs

in Java. It provides a set of data structures to represent trees and graphs and some graph drawing algorithms based on these data structures.

1.4 JDSL

We introduce version 2.0 of the *Data Structures Library in Java* (*JDSL*), which provides advanced data structures and algorithms not found in the Java Collections and JGL. In addition to basic data structures such as lists and dictionaries, JDSL includes implementations of a variety of complex data structures such as trees, graphs, and priority queues, with powerful and flexible access to the elements stored.

Besides providing iterators, a simple mechanism for iteratively listing through a collection of objects, JDSL introduces two new types of accessors to data, called *positions* and *locators*, which allow to "track" elements within data structures. For example, the method for inserting an element into a dictionary returns a locator for the element, which allows to later access the element in constant time without having to search.

Another feature of JDSL is the support for *decorations* (or *attributes*), which can be used to "label" positions within a data structure. For example, in a traversal of a graph, we can use decorations to mark the vertices and edges visited so far.

JDSL views algorithms as objects that (*i*) are instantiated with the input data, and (*ii*) provide access to the output after their execution via various methods. Algorithms within JDSL can be parameterized by means of the *template method pattern* [6].

JDSL supplements the Java Collections and is not meant to replace them. No conflicts arise when using in the same program data structures from JDSL and from the Java Collections. To facilitate the use of JDSL data structures in existing programs, adapter classes are provided to translate a collection

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

Table 1: A Comparison of the Java Collections, the Generic Library for Java, the Graph Foundation Classes for Java, and the Data Structures Library in Java.

into a JDSL container and back, whenever when such a translation is applicable.

Table 1 compares key features of JC, JGL, GFC, and JDSL. In the current JDSL 2.0 release, our main emphasis has been on data structures while only a basic repertory of algorithms has been provided. Future releases will include a wider collection of algorithms. Additional future work includes supporting full thread-safety and serializability within JDSL.

JDSL has been developed at the *Center for Geometric Computing* at Brown University and at Algomagic Technologies, Inc., in collaboration with Michael Goodrich, who is a professor at The Johns Hopkins University.

In the next section, we present the design goals for JDSL. Section 3 discusses the major concepts used in JDSL, and Section 4 examines the specific data structures and algorithms packaged in release 2.0 of JDSL. In Section 5, we briefly overview the history of the JDSL project. The software engineering practices applied during the project are discussed in Section 6. We finally consider future directions for JDSL in Section 7.

2 Design Goals

In this section, we overview the design goals followed in the development of JDSL.

2.1 Flexibility

JDSL was designed to be easily adaptable to a number of purposes. A first use of the JDSL data structures is to deploy them directly in a software application. Alternatively, one can create new data structures using the JDSL data structures as building blocks.

2.2 Reliability

Reliability has been one of our primary goals during the construction of JDSL. We have put considerable effort into designing exact specifications for every JDSL component, and we have created a suite of detailed exceptions that will be thrown in the event of bad input data, to notify the user of the error and shed light on the nature of the misuse. Finally, we have gone through an extensive and exhaustive set of reviews and tests, which are discussed in more detail in Section 6.

2.3 Efficiency

The JDSL data structures typically offer the best-possible asymptotic time complexity for every operation supported. While flexibility and reliability necessarily penalize (by a constant factor) the actual running time and space requirement, various performance improvement techniques have been adopted, including on-demand creation and caching of auxiliary structures.

2.4 Object-Orientation

JDSL takes a strong object-oriented view of data structures and algorithms. The JDSL data structures are objects that handle themselves all the supported operations. Algorithms too are objects in JDSL. Algorithm objects are instantiated with the input data. They store the results of their execution and provide methods to access them.

3 Data Organization Concepts in JDSL

In this section, we examine some key data organization concepts used in JDSL.

3.1 Container

JDSL views a data structure as an organized collection of objects, called the *elements* of the data structure. All data structures in JDSL implement the `Container` interface, which defines basic methods such as reporting the number of elements, and returning an iterator to the elements.

3.2 Element

An element of a JDSL data structure is any `java.lang.Object`. Note that the same object can be stored in many data structures and can be also stored multiple times in the same data structure. That is, two elements of different containers, or of the same container, can be the same object.

3.3 Key

Some data structures in JDSL, called *key-based containers*, store *keys* associated with elements. Keys are typically used as an indexing mechanism for their associated elements. An example of a key-based data structure is a *dictionary*, whose main methods support the following operations:

- inserting a (key, element) pair;
- searching for an element with a given key;
- removing an element with a given key.

A key can be any `java.lang.Object`. Note that a key and its element need not be distinct from each other. Typical keys are strings (e.g., names) and numbers (e.g., account numbers).

3.4 Accessor

JDSL provides unified and implementation-independent access to the elements of a data structure by means of *accessors* [8]. An accessor abstracts the notion of membership of an element into a container hiding the details of the implementation. An accessor provides constant-time access to an element stored in a data structure, independently from the implementation of the data structure. Every element has an accessor associated with it. Most operations in JDSL data structures refer to elements through their accessors.

For example, in JDSL a sequence `S` may be implemented either by means of an array or by means of a linked list. In the first case, to access an element we need its index in the array. In the second case, we need a pointer to the list node storing the element. However, the user of JDSL need not know which implementation of the sequence is being used since in either case, given an accessor `acc` to the sequence, we can get the element by calling `acc.element()`. Also, we can delete the element from the sequence by calling `S.remove(acc)`.

There are two types of accessors in JDSL, *positions* and *locators*, and they are used for different types of data structures. See Figure 1 for a diagram of the accessor interface hierarchy.

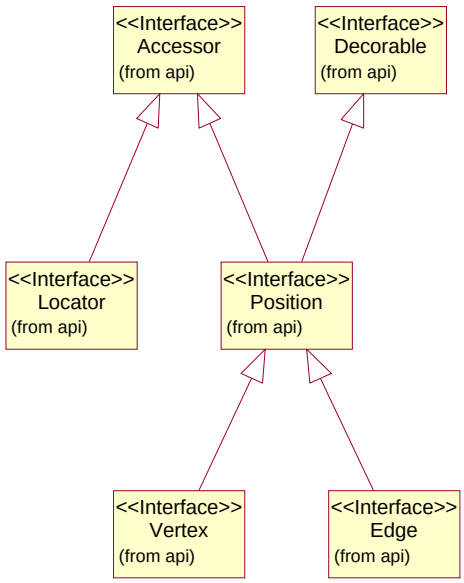

Figure 1: The Accessor interface hierarchy.

3.4.1 Position

Positions denote virtual "places" in data structures such as sequences, trees, and graphs, that can be modeled by a network. These data structures are called *positional containers* and establish topological relations between their positions, such as the adjacency relation on the vertices of a graph, the parent-child relation on the nodes of a tree, and the predecessor-successor relation on the nodes of a sequence. Positions represent the "nodes" in such structures.

Positions are similar to the concept of items used in the LEDA C++ library [12].

3.4.2 Locator

The accessors for key-based containers are called *locators*. Since key-based containers do not define per se a network of virtual places but operate on their elements through their associated keys, a locator is conceptually different from a position.

A locator is a coat-check, of sorts, for a (key,element) pair inside a key-based container `C`. If you give the container a pair to hold, the container gives you back a locator `loc`, and you can later refer to the pair by means of the locator. For example, you can get the key with `loc.key()`, you can remove the pair by calling `C.remove(loc)`, and you can change the existing element with a new element `newel` by calling `C.replaceElement(loc,newel)`.

While providing "positionless" interface methods, key-based container must be ultimately implemented with a concrete positional data structure where the (key, element) pairs may change positions due to internal restructurings. For example, a typical implementation of a priority queue uses a binary tree (heap) where the (key, element) pairs move around the tree to preserve the top-down ordering of the keys (up-heap and down-heap).

Hence 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.

3.5 Iterator

Iterators provide a simple mechanism for iteratively listing through a collection of objects. JDSL provides various iterators over the elements and the accessors of a data structure. See Figure 2 for a diagram of the iterator interface hierarchy.

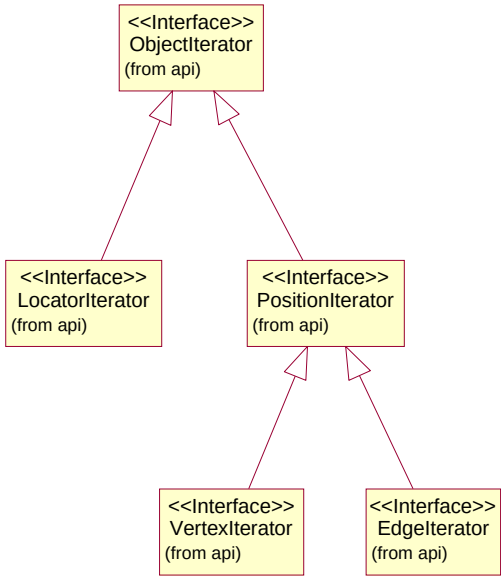

Figure 2: The Iterator interface hierarchy.

All the JDSL data structures provide methods that return iterators spanning the entire data structure (e.g., all the nodes of a tree or all the locators in a dictionary). In addition some methods provide access to subsets of the data structure (e.g., the children of a node of a tree or the locators with a given key). JDSL iterators can be traversed only forward. However, they can be reset to the beginning to repeat the traversal.

Iterators in JDSL have specific *snapshot* semantics: they refer to the state of the data structure at the time the iterator was created, even in the data structure has been modified while stepping through the iterator. For example, if an iterator is created for all the nodes of a tree and then a subtree is cut off, the iterator will still include the nodes of the removed subtree.

Note that the snapshot semantics refers only to the objects traversed by the iterator and not to their state (e.g., other objects referred to by them). For example, let `objiter` be an iterator over the elements stored in a sequence, and let `positer` be an iterator over the positions of the sequence. If after the creation of these iterators we change an element at a certain position, then `objiter` will still contain the old element but the position obtained from `positer` will refer to the new element — indeed, the position is still the same although its element changed.

3.6 Comparator

When using a key-based container, it is particularly important to be able to specify the relation for comparing the keys. In general, this relation depends on the type of the keys and on the specific application for which the key-based container is used. Keys of the same type may be compared differently in different applications.

To provide this capability, the `EqualityComparator`, `Comparator`, and `HashComparator` interfaces are defined in JDSL (see Figure 3). Interface `EqualityComparator` defines two methods: `isComparable(Object)`, for checking whether the object is a member of the ordered set over which the comparator is defined, and `isEqualTo(Object,Object)`, for testing whether the two objects are equal. Interface `Comparator` extends interface `EqualityComparator` with methods for testing whether the first object is less than, less than or equal to, greater than, greater than or equal to the second object, and with method `compare(Object,Object)`, a C-style comparison function. Interface `HashComparator` extends interface `EqualityComparator` with method `hashValue(Object)`, to be used in the hash table implementation of the `Dictionary` interface.

The concept of comparator is present also in the `java.util` package of Java 2 SDK, version 1.2, where a `Comparator` interface is defined. In order to maintain the backward compatibility of JDSL with Java JDK, version 1.1, the JDSL `Comparator` does not extend the Java `Comparator`.

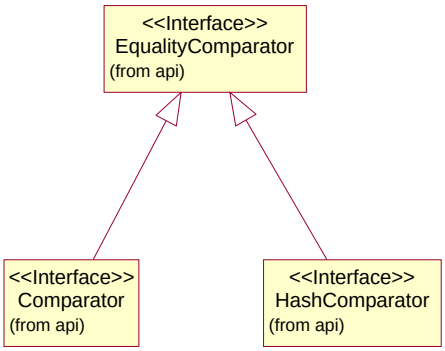

Figure 3: The Comparator interface hierarchy.

3.7 Decoration

Another feature of JDSL is the ability to "decorate" the positions of a data structure with attributes through the `Decorable` interface. This mechanism is useful for storing intermediate or final results of the execution of an algorithm. For example, in a depth-first search traversal of a graph, we can use decorations to mark the vertices being visited and to store the computed DFS number of each vertex.

3.8 Algorithm/Template Method Pattern

JDSL views algorithms as objects that are instantiated with the input data, and provide access to the output after their execution via various methods. Algorithms in JDSL perform "generic" computations that can be parameterized by means of the *template method pattern* [6]. In short, this means that they can be specialized for specific tasks by subclassing them and overriding specific methods.

For example, JDSL contains an implementation of Dijkstra's shortest path algorithm that refers to the edge weights by means of a abstract method that can be specialized depending on how the weights are actually stored or computed in the application at hand.

4 The Architecture of JDSL

In this section, we describe the interfaces defined in JDSL, their implementations, and the algorithms that operate on them. Each data structure, with the notable exceptions of the priority queue and the graph, is described by two interfaces:

- The first interface contains all the methods to query the data structure. Its name has the prefix `Inspectable`.
- The second interface, which extends the first one, contains all the methods to modify the data structure.

As described in Section 3, we can partition the set of data structures that are implemented in JDSL into two subsets: the positional containers and the key-based containers. Accordingly, the interfaces for the various containers are organized into two hierarchies (see Figures 4 and 5), with a common root given by interfaces `InspectableContainer` and `Container`. At the same time, interfaces, classes, and algorithms are grouped into various Java packages.

In the rest of this section, we denote with N the current number of elements stored in the data structure being considered.

4.1 Packages

The architecture of JDSL currently consists of eight Java packages. Each package consists of a set of interfaces and/or classes. Interfaces and exceptions for the data structures are defined in packages with the `api` suffix, while the reference implementations of these interfaces are defined in packages with the `ref` suffix. Interfaces, classes, and exceptions for the algorithms are instead grouped on a functional basis. As we will see later, the interfaces are arranged in hierarchies that may extend across different packages. The current packages are the following:

- `jdsl.core.api`. Package of interfaces and exceptions that compose the API for the core data structures of JDSL: sequences, trees, priority queues, dictionaries, and iterators on their elements, positions and locators.
- `jdsl.core.ref`. Package of implementations of the interfaces in `jdsl.core.api`. Most implementations have names of the form `<ImplementationStyle><InterfaceName>`. For instance, an `ArraySequence` implements the `jdsl.core.api.Sequence` interface with a growable array. 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`. Package of sorting algorithms that operate on `Sequence` objects defined in `jdsl.core.api`. They all implement the `SortObject` interface, also defined in this package.
- `jdsl.core.algo.traversals`. Package of traversal algorithms that operate on `jdsl.core.api.InspectableTree` objects. A traversal algorithm performs operations while visiting the nodes of the tree.
- `jdsl.core.util`. This package currently contains a `Converter` class to convert JDSL data structures to Java Collections and vice versa.
- `jdsl.graph.api`. Package of interfaces and exceptions that compose the API for the graph data structure of JDSL.

- `jdsl.graph.ref`. The package contains class `IncidenceListGraph`, an implementation of interface `jdsl.graph.api.Graph`, plus implementations of iterators over the vertices and edges of the graph.
- `jdsl.graph.algo`. Package of basic graph algorithms, including algorithms for depth-first search, topological sort, single-source shortest path, and minimum spanning tree.

4.2 Positional Containers and their Algorithms

Positional containers store elements, which can be accessed through the container's positions. All positional containers implement interfaces `InspectablePositionalContainer` and `PositionalContainer`, which extend `InspectableContainer` and `Container`, respectively (see Figure 4). Every positional container implements a set of essential operations, including being able to determine its own size (`size()`), to determine whether it contains a specific position (`contains(Accessor)`), to replace the element associated with a position (`replaceElement(Accessor,Object)`), to swap the elements associated with two positions (`swapElements(Position,Position)`) and to get iterators over the elements (`elements()`) or the positions (`positions()`) of the container.

4.2.1 Sequence

A sequence is a basic data structure used for storing elements in a linear, ranked fashion. Sequences can be implemented in many ways, e.g., as a linked list of nodes or on top of an array.

Sequence interfaces. In JDSL, sequences are described by interfaces `InspectableSequence` and `Sequence`, which extend `InspectablePositionalContainer` and `PositionalContainer`,respectively. In addition to the basic methods provided by all positional containers, the sequence interfaces support access and modification to positions at the sequence ends (with methods such as `first()`, `insertLast()`, and `removeFirst()`) and to specific positions along the sequence (with methods such as `after(Position)`, `atRank(int)`, `insertBefore(Position)`, and `removeAtRank(int)`).

NodeSequence. `NodeSequence` is an implementation of `Sequence` built on top of a doubly-linked list of nodes. The nodes are actually the positions of the sequence. `NodeSequence` takes $O(1)$ time to insert, remove, or access both ends of the sequence or a position before or after a given one. It takes $O(N)$ time to insert, remove, or access positions in terms of their rank in the sequence, since it must traverse along the sequence to find a position's rank. Some appropriate uses of a `NodeSequence` include using it as a stack, a queue, or a deque.

ArraySequence. `ArraySequence` is an implementation of `Sequence` built on top of a growable array of positions. It can be created with an initial capacity, and can be told whether or not to reduce this capacity when its size drops below a certain value, depending on whether the user prefers space or speed efficiency. `ArraySequence` takes $O(1)$ time to access any position in the sequence, and $O(1)$ time (amortized))to insert or remove at the end of the sequence over a series of calls. It takes $O(N)$ time to insert or remove at the beginning or middle of the sequence, since elements have to be shifted to make room for the new position or to close up the space left by the removed position. Hence, an `ArraySequence` is useful for quick access after filling the sequence, for filling the sequence only at the end, or for using it as a stack.

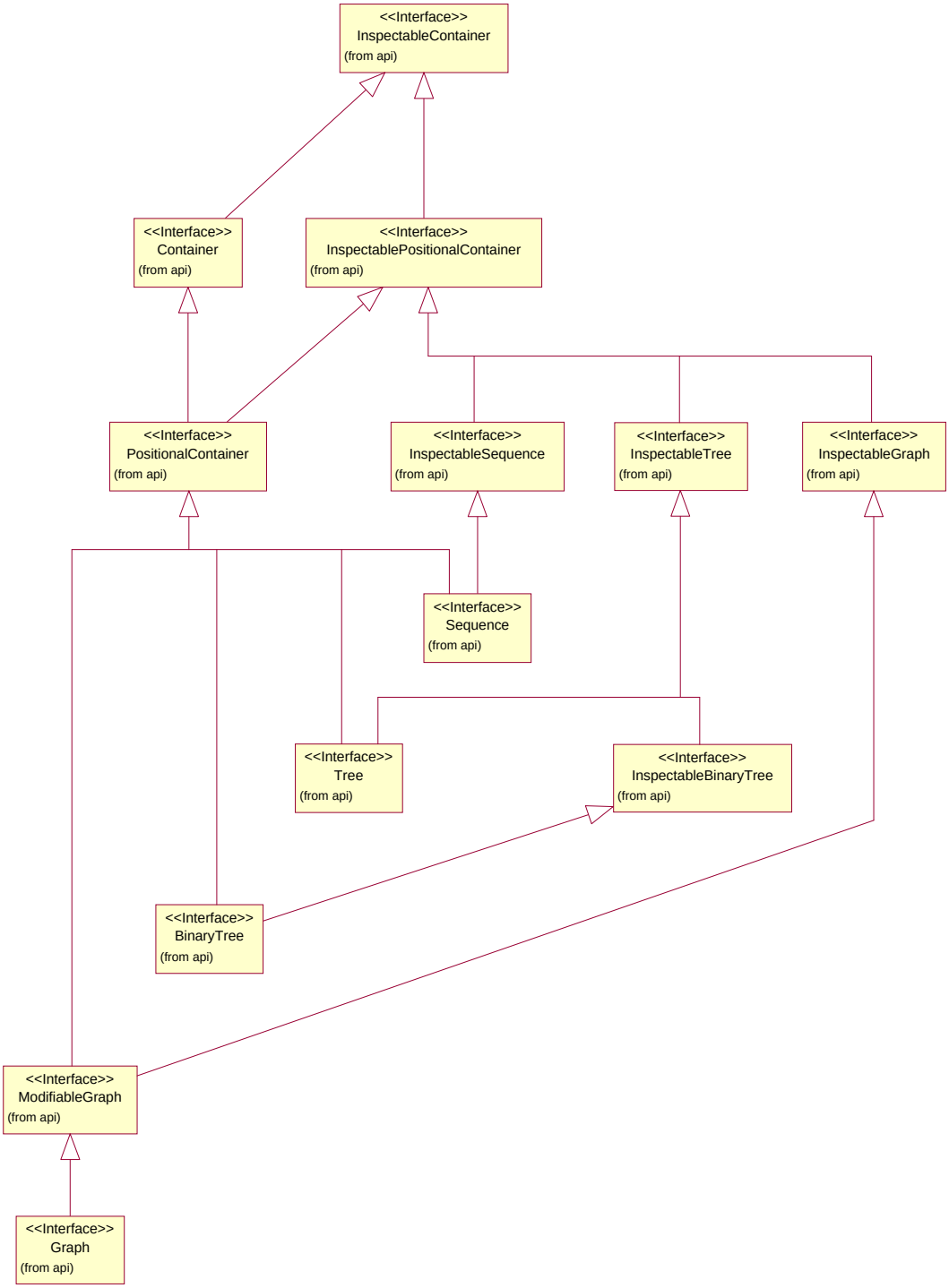

Figure 4: The PositionalContainer interface hierarchy.

Sorting algorithms. JDSL provides a suite of sorting algorithms for different applications. They all implement the `SortObject` interface, whose only method is `sort(Sequence,Comparator)`. Sorts with the prefix `List` are most efficient when used with `NodeSequence`'s and sorts with the prefix `Array` are most efficient when used with `ArraySequence`'s.

- `ListQuickSort/ArrayQuickSort.` Quicksort is an extremely fast sort, running in $O(N \log N)$ expected time. However, its performance degrades greatly if the sequence is already very close to being sorted. Also, it is not *stable* — that is, it does not guarantee that elements with the same value will remain in the same order they were in before sorting. In all cases whether neither of these caveats apply, it is the best choice.
- `ListMergeSort/ArrayMergeSort.` Mergesort is not as fast as quicksort, though it still runs in $O(N \log N)$ time. There are no cases where its performance will degrade due to peculiarities in the input data, and it is a stable sort.
- `HeapSort.` Heapsort is another sort, which is based upon a `ArrayHeap` (see below). Its performance, like that of mergesort, will not degrade due to peculiarities in the input data, but it is not a stable sort.

4.2.2 Tree

Trees allow more sophisticated relationships between elements than is possible with a sequence. Trees allow relationships between a child and its parent, or between siblings of one parent. Trees have one root, which has no parent, and external leaves, which have no children.

Tree interfaces. `InspectableTree` and `Tree` are the interfaces describing an n-ary tree; they extend `InspectablePositionalContainer` and `Container`, respectively. `InspectableBinaryTree`, which extends `InspectableTree`, and `BinaryTree`, which extends `Container`, are the interfaces describing a binary tree. The nodes of the tree are the positions of these containers. In addition to the basic methods provided by all positional containers, the tree interfaces define methods to determine where in the tree a position lies (with methods such as `isRoot(Position)` and `isExternal(Position)`), to return the parent (`parent(Position)`), siblings (`siblings(Position)`), or children (with methods such as `children(Position)`, `leftChild(Position)`, and `childAtRank(Position,int)`) of a position, to cut (`cut(Position)`) or link (`link(Position,Tree)`) a subtree, etc.

NodeTree. `NodeTree` is an implementation of `Tree`. It always contains at least one node. It is the class to use when a generic tree is needed, or for building more specialized (non-binary) types of trees.

NodeBinaryTree. `NodeBinaryTree` is an implementation of `BinaryTree`. Similarly to `NodeTree`, it always contains at least one node; in addition, each node can have either 0 or 2 children. If a more complex tree is not necessary, using a `NodeBinaryTree` will be faster and easier than using a `NodeTree`.

Iterator-based traversals. In JDSL, there are two different types of traversals for trees. The first type is based on iterators. For an iterator, the tree to iterate over must be passed in at instantiation. After instantiation, the tree can be iterated over using methods `nextPosition()` and `reset()`. Iterators have snapshots semantics — see Section 3.5. Iterators give a quick traversal of the tree in a specific order, and are the proper traversal to use when this is all that is required.

- `PreOrderIterator.` A preorder iterator gives its traversal in pre-order, that is, it returns a parent before returning any of its children. Preorder iterators work both for general and binary trees.

- `PostOrderIterator`. A postorder iterator gives its traversal in post-order, which means that it returns a parent after returning all of its children. Postorder iterators work both for general and binary trees.
- `InOrderIterator`. An inorder iterator gives its traversal in in-order, which means that it returns a parent in between its left and right children. Inorder iterators only work for binary trees.

Algorithmic traversals. The second type of tree traversal in JDSL uses an algorithm object, which can be extended through the template method pattern.

- `EulerTour`. The only traversal algorithm object is the Euler tour, which visits each node several times. Namely, a first time before traversing any of the subtrees of the node, between the traversals of any two consecutive subtrees, and a last time after traversing all the subtrees. Each time a node is visited, one of the methods `visitFirstTime(Position)`, `visitBetweenChildren(Position)`, `visitLastTime(Position)`, and `visitExternal(Position)` of this algorithm object is automatically invoked. A particular computation on the visited tree may be performed by suitably overriding these methods in a subclass of `EulerTour`.

  A Euler tour should be used rather than an iterator if the traversal is to be used in a more sophisticated manner than iterators allow. Note that, unlike the iterators, the Euler tour does not have snapshot semantics. This means that any modification of the tree during execution of the Euler tour will cause undefined behavior.

4.2.3 Graph

A graph is a fundamental data structure used in a variety of application areas describing a binary relationship on a set of elements. Each vertex of the graph may be linked to other vertices through edges. Edges can be either one-way, *directed* edges, or two-way, *undirected* edges. In JDSL, both vertices and edges are positions of the graph. JDSL handles all graph special cases such as self-loops, multiple edges between two vertices, and disconnected graphs.

Graph interfaces. The main graph interfaces are `InspectableGraph`, which extends `InspectablePositionalContainer`, `ModifiableGraph`, which extends `PositionalContainer`, and `Graph`, which extends both `InspectableGraph` and `ModifiableGraph`. The methods in these interfaces allow to:

- determine whether vertices are adjacent (`areAdjacent(Vertex,Vertex)`) or vertices and edges are incident (`areIncident(Vertex,Edge)`);
- determine the degree of a vertex (`degree(Vertex)`);
- determine the origin (`origin(Edge)`) or destination (`destination(Edge)`) of an edge.
- add (`insertVertex(Object)`) or remove (`removeVertex(Vertex)`) vertices;
- set the direction of an edge (`setDirectionFrom(Edge,Vertex)` and `setDirectionTo(Edge,Vertex)`);
- add (`insertEdge(Vertex,Vertex,Object)`), delete (`removeEdge(Edge)`), split (`splitEdge(Edge,Object)`), or unsplit (`unsplitEdge(Vertex,Object)`) edges.

IncidenceListGraph. `IncidenceListGraph` is an implementation of `Graph`. It is based on an incidence list representation of a graph.

Traversals. The depth-first search (DFS) traversal of a graph is available in JDSL. Depth-first search proceeds along one path, continuing until no new vertices can be found before backtracking. The JDSL implementation of depth-first search is a template method that allows the user to specify actions to occur when a vertex is first visited or is "finished" by being exited for the last time, and when different sorts of edges are reached (such as tree edges in the search tree DFS generates, or cross edges between different branches of the search tree). The basic implementation of depth-first search is designed to work on undirected graphs.

Two subclasses of `DFS` are provided: `DirectedDFS` (for directed graphs) and `DirectedFindCycleDFS`, which finds cycles. They are good examples of how to use `DFS` as a template method to implement a more specific algorithm.

Topological numbering. A topological numbering is a numbering of the vertices of an acyclic directed graph such that, if there is an edge from vertex u to vertex v, then the number associated with v is higher than the number associated with u.

Two algorithms that compute a topological numbering are included in JDSL — `TopologicalSort` associates with each vertex a unique number, whereas `UnitWeightedTopologicalNumbering` associates with each vertex a number based on how far it is from the source of the graph, and vertices may have equal numbers if they are at the same distance from the source. Both topological numbering algorithms extend the abstract class `AbstractTopologicalSort`.

Dijkstra's algorithm. Dijkstra's algorithm computes the shortest path to every vertex of a connected graph (with weights assigned to the edges by a function), from a specific source vertex. The JDSL implementation of Dijkstra's algorithm — `IntegerDijkstraTemplate` — uses the template method pattern; it can be easily extended to change its functionality. Extending it makes it possible, for instance, to stop after computing the shortest path to a specific vertex, to alter the function for calculating the weight of an edge, and to change the way the results are stored.

Prim's algorithm. Prim's algorithm computes a minimum spanning tree of a graph (the shortest set of paths connecting every vertex to every other vertex). The JDSL implementation of Prim's algorithm — `IntegerPrimTemplate` — uses the template method pattern; it can be easily extended to change its functionality. Extending it makes it possible, for instance, to stop after computing the tree for a limited set of vertices, to set the function for calculating the weight of an edge, and to change the way the results are stored.

4.3 KeyBased Containers and their Algorithms

Key-based containers store key-element pairs, which can be accessed through the container's locators. All key-based containers implement interfaces `InspectableKeyBasedContainer` and `KeyBasedContainer`, which extend `InspectableContainer` and `Container`, respectively (see Figure 5). Every key-based container implements a set of essential operations, including being able to determine its own size (`size()`), to determine whether it contains a specific locator (`contains(Accessor)`), to replace the key (`replaceKey(Locator,Object)`) or the element (`replaceElement(Accessor,Object)`) associated with a locator, to insert (`insert(Object,Object)`) or remove (`remove(Locator)`) a key-element pair, and to get iterators of the locators (`locators()`), keys (`keys()`), or elements (`elements()`) in the container.

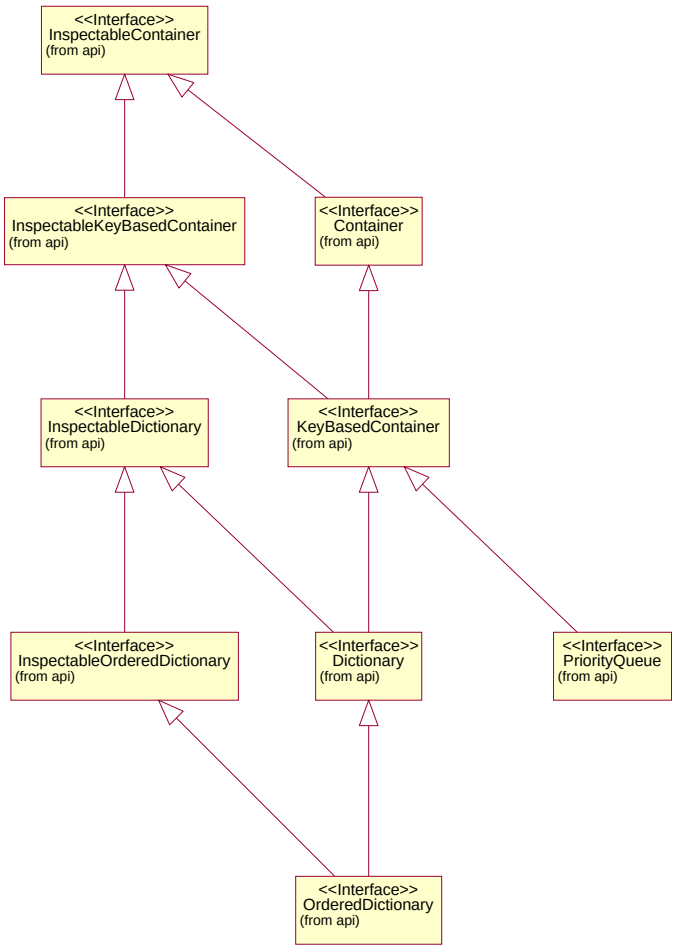

Figure 5: The KeyBasedContainer interface hierarchy.

4.3.1 PriorityQueue

A priority queue is a data structure for storing a collection of elements prioritized by keys, where the smallest key value indicates the highest priority. It supports arbitrary insertions and deletions of elements and keeps track of the highest-priority key. A priority queue is useful, for instance, in applications where the user wishes to store a queue of tasks of varying priority, and always process the most important task next.

PriorityQueue interface. The `PriorityQueue` interface extends the `KeyBasedContainer` interface. In addition to the basic methods common to all the key-based containers, it provides methods to access (`min()`) or remove (`removeMin()`) the key-element pair with highest priority, i.e., with minimum key. Note that an element's priority can be changed by method `replaceKey(Locator,Object)`, inherited from `KeyBasedContainer`.

ArrayHeap. `ArrayHeap` is an efficient implementation of `PriorityQueue` built upon a heap. Inserting, removing, or changing the key of a key-element pair takes logarithmic time, and examining the key-element pair with the minimum key can be done in constant time. The implementation is parameterized with respect to the comparison rule used to order the keys; to this purpose, a `Comparator` object is passed as an argument to the `ArrayHeap` constructors.

4.3.2 Dictionary

A dictionary is a data structure used to store key-element pairs and then quickly search for them using their keys. An ordered dictionary is a particular dictionary where a total order on the set of keys is defined. All JDSL dictionaries are multi-maps, which means that they can store multiple key-element pairs with the same key.

Dictionary interfaces. The primary dictionary interfaces are `InspectableDictionary` and `Dictionary`, which extend `InspectableKeyBasedContainer` and `KeyBasedContainer`, respectively. In addition to the basic methods common to all the key-based containers, these dictionary interfaces provide methods to find key-element pairs by their keys (`find(Object)` and `findAll(Object)`) and to remove all key-element pairs with a specific key (`removeAll(Object)`). Other dictionary interfaces are `InspectableOrderedDictionary` and `OrderedDictionary`, which extend `InspectableDictionary` and `Dictionary`, respectively. They provide additional methods to access the first (`first()`) or last (`last()`) key-element pair in the ordered dictionary, and to access the key-element pair before (`before(Locator)`) or after (`after(Locator)`) a given key-element pair.

HashtableDictionary. `HashtableDictionary` is an implementation of `Dictionary`. It is, as its name implies, a dictionary built on top of a hash table. Insertion and removal of key-element pairs will usually take $O(1)$ time, although individual insertions and removals may require $O(N)$ time. The implementation is parameterized with respect to the hashing function used to store the key-element pairs; to this purpose, a `HashComparator` object is passed as an argument to the `HashtableDictionary` constructors. `HashtableDictionary` is a good choice when overall speed is necessary.

RedBlackTree. `RedBlackTree` is an implementation of `OrderedDictionary`. It is a particular type of binary search tree, where insertion, removal, and access to key-element pairs require each $O(\log N)$ time. The implementation is parameterized with respect to the comparison rule used to order the keys; to this purpose, a `Comparator` object is passed as an argument to the `RedBlackTree` constructors.

5 Project History

The initial development of JDSL began in September 1996. A major part of the project in the first years was the experimentation with different models for data structures and algorithms, and the construction of prototypes.

In 1997 and 1998, early prototypes of JDSL were used in the introductory data structures and algorithms course at Brown University. After a successful class experimentation, JDSL 1.0, a first version of JDSL aimed at instructional applications, was released in 1998. JDSL 1.0 also included auxiliary packages for visualizing data structures and for testing their implementation [2].

In the Spring of 1999, a new experimental version of JDSL (more advanced than 1.0) was used in programming assignments and course projects in a graduate-level computational geometry course at Brown University.

A significant implementation, documentation, and testing effort was done in the Summer of 1999, leading to the current version of JDSL, JDSL 2.0, which was released in August 2000 after a complete redesign of the JDSL Web site. JDSL 2.0 is suitable for use by researchers, professionals, educators, and students. It is licensed free of charge for educational and research purposes.

The two releases of JDSL were accompanied by the publication of the book *Data Structures and Algorithms in JAVA* by Goodrich and Tamassia [9, 10].

6 Software Engineering in JDSL

In any project of the size and scope of JDSL (over 30,000 lines of code and internal documentation in the current version), careful attention must be paid to good software engineering practice and quality control. In coming up with our process for designing, implementing, and verifying our data structures, we looked with interest at the procedures followed in the construction of previous software libraries, as well as those commonly in use in the industry.

6.1 Design process

In the creation of the most recent version of JDSL, we followed an engineering plan consisting of several distinct steps.

Interface/Specifications design. The first step was to design the interfaces and specifications that we would use to construct our data structures and algorithms. This took place in several design meetings. Discussion in general continued on an email discussion list, and often decisions would be considered multiple times before being finalized. Occasionally, a prototype implementation would be constructed in order to see firsthand the advantages and disadvantages of a particular interface choice. Interface and specification decisions and clarifications continued apace during the project, and deciding when to "freeze" an interface from further changes was an ongoing challenge.

Implementation. After a satisfactory interface was designed for the particular data structure or algorithm, it would be implemented by a member of the research team. Occasionally, a flaw in the specification discovered through implementation would necessitate a return to the previous stage in development.

Documentation. Immediately after the implementation of a data structure or algorithm, the implementor's next task would be to fully document the class(es) they had just constructed. We chose the Javadoc standard for our commenting [18]. In addition, we required that every method's time complexity be commented.

Testing. Testing of our data structures and algorithms took place in two stages. First, during implementation, the programmer would conduct a "white-box" test to see if every method functioned correctly, with special attention to implementation-specific details. For example, for our `RedBlackTree`, this test included verifying that an extensive set of insertions and deletions did not compromise the black-height or double-red properties.

The second stage was a "black-box" test, conducted by an individual other than the implementor. This test examined every interface method of the data structure to verify that it functioned exactly as written in the specifications, both in exceptional and nominal cases. This test was crucial, as it was especially effective at catching inconsistencies created by late changes in specifications.

Reviews. We chose to emulate the common industry model of having two different sorts of reviews of code: peer reviews, and project leader reviews. During reviews, the reviewer would examine the code for correctness (though problems in this area were quite rare by this point), efficiency, and readability.

Peer reviews were performed by another member of the development team. Some peer reviews took place face-to-face, while in some other cases the reviewer examined the class on his or her own and sent comments and questions by email. Recommendations given during peer reviews were not mandatory, but were usually followed.

Project leader reviews were conducted by one of the project leaders, and were always face-to-face. Since this was the final review stage, reviews were extremely methodical. Any changes recommended by the project leaders were mandatory, and occasionally a second project leader review would take place if the necessary changes were sufficiently comprehensive.

6.2 Software tools

We used an extensive set of software tools to assist us in constructing JDSL. However, a few merit special mention.

For testing data structures, we used a testing package that we developed in-house [2]. This package gave us the capability to quickly generate data structures to compare to one another, and allowed us to test methods for correct return, for leaving the structure in the correct state, and for correctly throwing exceptions on erroneous input.

With over 20 developers, some of whom working remotely, we found that a system to maintain a common code repository and to manage code revisions was essential. We adopted CVS, the Concurrent Versioning System [3], which performed quite well.

7 Future Work

Several extensions are planned for future versions of JDSL, including:

- additional graph algorithms, such as breadth-first search, maximum flow, and matching;
- data structures and algorithms for geometric computing (the GeomLib project [19]), such as planar subdivisions, Voronoi diagrams, and convex hulls;
- data structures and algorithms for graph drawing, including straight-line, hierarchical, and orthogonal layouts [7];
- an instructional tool for the visualization of basic data structures (the JDSLViz project [2]);
- a package that facilitates testing whether the implementation of a data structure complies with the interface specification [2].

8 Acknowledgments

We would like to acknowledge suggestions, ideas, and code prototypes given by James Baker, Don Blaheta, Lubomir Bourdev, Jitchaya Buranahirun, Ming En Cho, Marco da Silva, John Kloss, David Jackson, Masi Oka, and Amit Sobti. Finally, we would like to thank Franco Preparata for encouragement and support.

References

[1] alphaWorks. Graph Foundation Classes for Java.
`http://www.alphaworks.ibm.com/tech/gfc`.

[2] R. S. Baker, M. Boilen, M. T. Goodrich, R. Tamassia, and B. A. Stibel. Testers and visualizers for teaching data structures. In *Proc. 30th ACM SIGCSE Tech. Sympos.*, pages 261–265, 1999.

[3] P. Cederqvist. *Version Management with CVS*.
`http://www.loria.fr/~molli/cvs/doc/cvs.ps`.

[4] G. Di Battista, W. Didimo, A. Leonforte, M. Patrignani, and M. Pizzonia. GDToolkit.
`http://www.dia.uniroma3.it/~gdt/`.

[5] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL a computational geometry algorithms library. *Softw. – Pract. Exp.*, 30(11):1167–1202, 2000.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

[7] N. Gelfand and R. Tamassia. Algorithmic patterns for orthogonal graph drawing. In S. H. Whitesides, editor, *Graph Drawing (Proc. GD '98)*, volume 1547 of *Lecture Notes Comput. Sci.*, pages 138–152. Springer-Verlag, 1998.

[8] M. T. Goodrich, M. Handy, B. Hudson, and R. Tamassia. Accessing the internal organization of data structures in the JDSL library. In M. T. Goodrich and C. C. McGeoch, editors, *Algorithm Engineering and Experimentation (Proc. ALENEX '99)*, volume 1619 of *Lecture Notes Comput. Sci.*, pages 124–139. Springer-Verlag, 1999.

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

[10] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, New York, NY, 2nd edition, 2001.

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

[12] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, England, 1999.

[13] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, Reading, MA, 1996.

[14] P. Mutzel, C. Gutwenger, R. Brockenauer, S. Fialko, G. Klau, M. Krüger, T. Ziegler, S. Näher, D. Alberts, D. Ambras, G. Koch, M. Jünger, C. Buchheim, and S. Leipert. AGD, a library of Algorithms for Graph Drawing.
`http://www.mpi-sb.mpg.de/AGD/`.

[15] ObjectSpace. JGL, the Generic Collection Library for Java.
`http://www.objectspace.com/products/jgl`.

[16] J. Siek, L.-Q. Lee, and A. Lumsdaine. The generic graph component library. In *OOPSLA '99 Technical Notes*, 1999.

[17] Sun. Java 2 SDK, Standard Edition.
`http://java.sun.com/products/jdk/1.2`.

[18] Sun. Javadoc home page.
`http://java.sun.com/products/jdk/javadoc`.

[19] R. Tamassia and L. Vismara. A case study in algorithm engineering for geometric computing. *Internat. J. Comput. Geom. Appl.*, to appear.
`ftp://ftp.cs.brown.edu/pub/techreports/97/cs97-18.ps.Z`.