43

Data Structures in JDSL

Michael T. Goodrich
University of California, Irvine

Roberto Tamassia
Brown University

Luca Vismara
Brown University

43.1 Introduction

In the last four decades the role of computers has dramatically changed: once mainly used as *number processors* to perform fast numerical computations, they have gradually evolved into *information processors*, used to store, analyze, search, transfer, and update large collections of structured information. In order 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, computer programs need advanced *data structures* and *algorithms*. Implementing advanced data structures and algorithms, however, is not an easy task and presents some risks:

Complexity Advanced data structures and algorithms are often difficult to understand thoroughly and to implement.

Unreliability Because of their complexity, the implementation of advanced data structures and algorithms is prone to subtle errors in boundary cases, which may require a considerable effort to identify and correct.

Long development time As a consequence, implementing and testing advanced data structures and algorithms is usually a time consuming process.

As a result, programmers tend to ignore advanced data structures and algorithms and to resort to simple ones, which are easier to implement and test but that are usually not as efficient. It is thus clear how the development of complex software applications, in particular their rapid prototyping, can greatly benefit from the availability of libraries of reliable and efficient data structures and algorithms.

Various libraries are available for the C++ programming language. They include the *Standard Template Library* (STL, see Chapter 42) [9], now part of the C++ standard, the

1-58488-435-5/05/$0.00+$1.50

extensive *Library of Efficient Data Structures and Algorithms* (LEDA, see Chapter 41) [8], and the *Computational Geometry Algorithms Library* (CGAL) [3].

The situation for the Java programming language is the following: a small library of data structures, usually referred to as *Java Collections* (JC), is included in the java.util package of the Java 2 Platform.* An alternative to the Java Collections are the *Java Generic Libraries* (JGL) by Recursion Software, Inc.,† which are patterned after STL. Both the Java Collections and JGL provide implementations of basic data structures such as sequences, sets, maps, and dictionaries. JGL also provides a considerable number of generic programming algorithms for transforming, permuting, and filtering data.

None of the above libraries for Java, however, seems to provide a coherent framework, capable of accommodating both elementary and advanced data structures and algorithms, as required in the development of complex software applications. This circumstance motivated the development of the *Data Structures Library in Java* (JDSL) [10], a collection of Java interfaces and classes implementing fundamental data structures and algorithms, such as:

- sequences and trees;
- priority queues, binary search trees, and hash tables;
- graphs;
- sorting and traversal algorithms;
- topological numbering, shortest path, and minimum spanning tree.

JDSL is suitable for use by researchers, professional programmers, educators, and students. It comes with extensive documentation, including detailed Javadoc,‡ an overview, a tutorial with seven lessons, and several associated research papers. It is available free of charge for noncommercial use at `http://www.jdsl.org/`.

The development of JDSL began in September 1996 at the Center for Geometric Computing at Brown University and culminated with the release of version 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 [1] effort was carried out in 1999 leading to version 2.0, which was officially released in 2000. Starting with version 2.1, released in 2003 under a new license, the source code has been included in the distribution. During its life cycle JDSL 2.0 was downloaded by more than 5,700 users, while JDSL 2.1 has been downloaded by more than 3,500 users as of this writing. During these seven years a total of 25 people§ have been involved, at various levels, in the design and development of the library. JDSL has been used in data structures and algorithms courses worldwide as well as in two data structures and algorithms textbooks¶ written by the first two authors [6, 7].

In the development of JDSL we tried to learn from other approaches and to progress on them in terms of ease of use and modern design. The library was designed with the following goals in mind:

Functionality The library should provide a significant collection of existing data structures and algorithms.

*`http://java.sun.com/j2se/`

†`http://www.recursionsw.com/jgl.htm`

‡`http://java.sun.com/j2se/javadoc/`

§`http://www.jdsl.org/team.html`

¶`http://java.datastructures.net/` and `http://algorithmdesign.net/`

Reliability Data structures and algorithms should be correctly implemented, with particular attention to boundary cases and degeneracies. All input data should be validated and, where necessary, rejected by means of exceptions.

Efficiency The implementations of the data structures and algorithms should match their theoretical asymptotic time and space complexity; constant factors, however, should also be considered when evaluating efficiency.

Flexibility Multiple implementations of data structures and algorithms should be provided, so that the user can experiment and choose the most appropriate implementation, in terms of time or space complexity, for the application at hand. It should also be possible for the user to easily extend the library with additional data structures and algorithms, potentially based on the existing ones.

Observe that there exist some trade-offs between these design goals, e.g., between efficiency and reliability, or between efficiency and flexibility.

In JDSL 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 allows different implementations of the same interface to be used interchangeably, without having to modify the algorithm code.

A comparison of the key features of the Java Collections, JGL, and JDSL is shown in Table 43.1. The main advantages of JDSL are the definition of a large set of data structure APIs (including binary tree, general tree, priority queue and graph) in terms of Java interfaces, the availability of reliable and efficient implementations of those APIs, and the presence of some fundamental graph algorithms. Note, in particular, that the Java Collections do not include trees, priority queues and graphs, and provide only sorting algorithms.

	JC	JGL	JDSL
Sequences (lists, vectors)	✓	✓	✓
Trees			✓
Priority queues (heaps)		✓	✓
Dictionaries (hash tables, red-black trees)	✓		✓
Sets	✓		
Graphs			✓
Templated algorithms			✓
Sorting	✓	✓	✓
Data transforming, permuting, and filtering		✓	
Graph traversals			✓
Topological numbering			✓
Shortest path			✓
Minimum spanning tree			✓
Accessors (positions and locators)			✓
Iterators	✓	✓	✓
Range views	✓	✓	
Decorations (attributes)			✓
Thread-safety	✓		
Serializability	✓	✓	

TABLE 43.1 A comparison of the Java Collections (JC), the Generic Library for Java (JGL), and the Data Structures Library in Java (JDSL).

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 Java Collections and is not meant to replace them. No conflicts arise when using data structures from JDSL and from the Java Collections in the same program. To facilitate the use of JDSL data structures in existing programs, adapter classes are provided to translate a Java Collections data structure into a JDSL one and vice versa, whenever such a translation is applicable.

43.2 Design Concepts in JDSL

In this section we examine some data organization concepts and algorithmic patterns that are particularly important in the design of JDSL.

43.2.1 Containers and Accessors

In JDSL each data structure is viewed as a *container*, i.e., an organized collection of objects, called the *elements* of the container. An element can be stored in many containers at the same time and multiple times in the same container. JDSL containers can store heterogeneous elements, i.e., instances of different classes.‖

JDSL provides two general and implementation-independent ways to access (but not modify) the elements stored in a container: individually, by means of accessors, and globally, by means of iterators (see Section 43.2.2). An *accessor* [5] 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.

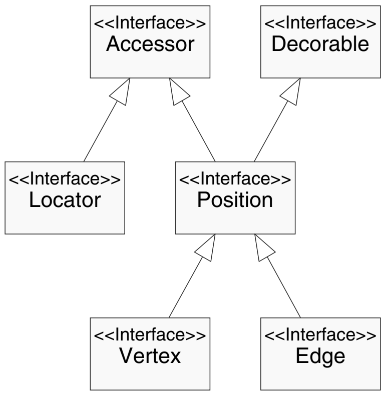


FIGURE 43.1: The accessors interface hierarchy.

We distinguish between two kinds of containers and, accordingly, of accessors (see Figure 43.1 for a diagram of the accessor interface hierarchy):

Positional containers Typical examples are sequences, trees, and graphs. In a positional container, some topological relation is established among the "place-

‖This is possible since in Java every class extends (directly or indirectly) java.lang.Object.

holders" 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. It is the user who decides, when inserting an element in the container, what the relationship is between the new "placeholder" and the existing ones (in a sequence, for instance, the user may decide to insert an element before a given "placeholder"). A positional container does not change its topology, unless the user requests a change specifically. 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 the cells of an array). Each position stores an element. Position implementations may store the following additional information:

- the adjacent positions (e.g., 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);
- consistency information (e.g., what container the position is in, the number of children in a tree).

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 an array cell. Through the positional container, instead, it is possible to replace the element of a position or to swap the elements between two positions. Note that, 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 [8].

Key-based containers Typical examples are dictionaries and priority queues. Every element stored in a key-based container has a *key* associated with it. Keys are used as an indexing mechanism for their associated elements. Typically, a key-based container is internally implemented using a positional container; for example, a possible implementation of a priority queue uses a binary tree (a heap). The details of the internal representation, however, are completely hidden to the user. Thus, the user has 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 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.

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 the element of a locator. An example of using locators is given in Section 43.4.

43.2.2 Iterators

While accessors allow users to 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, the keys (where present), and the positions or the locators of a container (see Figure 43.2 for a diagram of the iterator interface hierarchy). They are similar to the iterators provided by the Java Collections.

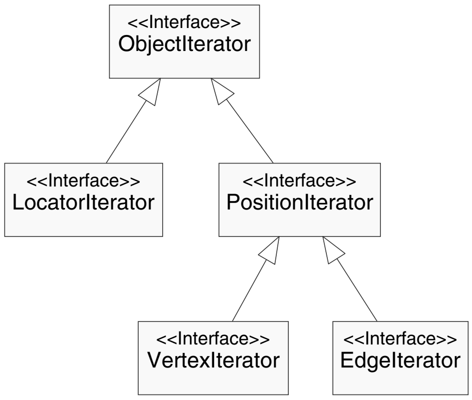


FIGURE 43.2: The iterators interface hierarchy.

All JDSL containers provide methods that return iterators over the entire container (e.g., all the positions of a tree or all the locators of a dictionary). In addition, some methods return iterators over portions of the container (e.g., the children of a position of a tree or the 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 reasons iterators in JDSL 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 positions of a tree and then a subtree is cut off, the iterator will still include the positions of the removed subtree.

43.2.3 Decorations

Another feature of JDSL is the possibility to "decorate" individual positions of a positional container with attributes, i.e., with 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 (DFS) traversal of a graph, we can use decorations to (temporarily) mark the vertices being visited and to (permanently) store the computed DFS number of each vertex. An example of using decorations is given in Section 43.4.

43.2.4 Comparators

When using a key-based container, the user 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 on the 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 fulfill this requirement is to specify the comparison relation through a *comparator* object, which is passed to the key-based container constructor and is then used by the key-based container every time two keys need to be compared.

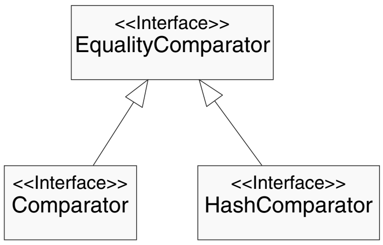


FIGURE 43.3: The comparators interface hierarchy.

Three comparator interfaces are defined in JDSL (see Figure 43.3 for a diagram of the comparators interface hierarchy). The concept of comparator is present also in the java.util package of the Java 2 Platform, where a Comparator interface is defined.

43.2.5 Algorithms

JDSL views algorithms as objects that receive the input data as arguments of their execute(.) method, and provide access to the output during or after the execution via additional methods. Most algorithms in JDSL are implemented following the *template method pattern* [4]. The invariant part of the algorithm is implemented once in an abstract class, deferring the implementation of the steps that can vary to subclasses. These varying steps are 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 for specific tasks by subclasses.

An example of applying the template method pattern is given in Section 43.4, where we use the JDSL implementation of Dijkstra's single-source shortest path algorithm [2]. 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.

43.3 The Architecture of JDSL

In this section we describe the interfaces of the data structures present in JDSL, the classes that implement those interfaces, and the algorithms that operate on them. Most containers are described by two interfaces, one (whose name is prefixed with Inspectable) that comprise all the methods to query the container, and the other, extending the first, that comprise all the methods to modify the container. Inspectable interfaces can be used as variable or argument types in order to obtain an immutable view of a container (for instance, to prevent an algorithm from modifying the container it operates on).

As described in Section 43.2.1, we can partition the set of containers present 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 43.4 and 43.5), with a common root given by interfaces InspectableContainer and Container. At the same time, container interfaces, their implementations, and algorithms that operate on them are grouped into various Java packages.

In the rest of this section, we denote with n the current number of elements stored in the container being considered.

43.3.1 Packages

JDSL currently consists of eight Java packages, each containing 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 Interfaces and exceptions that compose the API for the core containers (sequences, trees, priority queues, and dictionaries), for their accessors and comparators, 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 *ImplementationStyleInterfaceName*. 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 by applying the template method pattern.

jdsl.core.util This package contains a Converter class to convert some JDSL containers to the equivalent data structures of the 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 numbering, shortest path, and minimum spanning tree, all of which can be extended by applying the template method pattern.

43.3.2 Positional Containers

All positional containers implement interfaces InspectablePositionalContainer and PositionalContainer, which extend InspectableContainer and Container, respectively (see Figure 43.4).

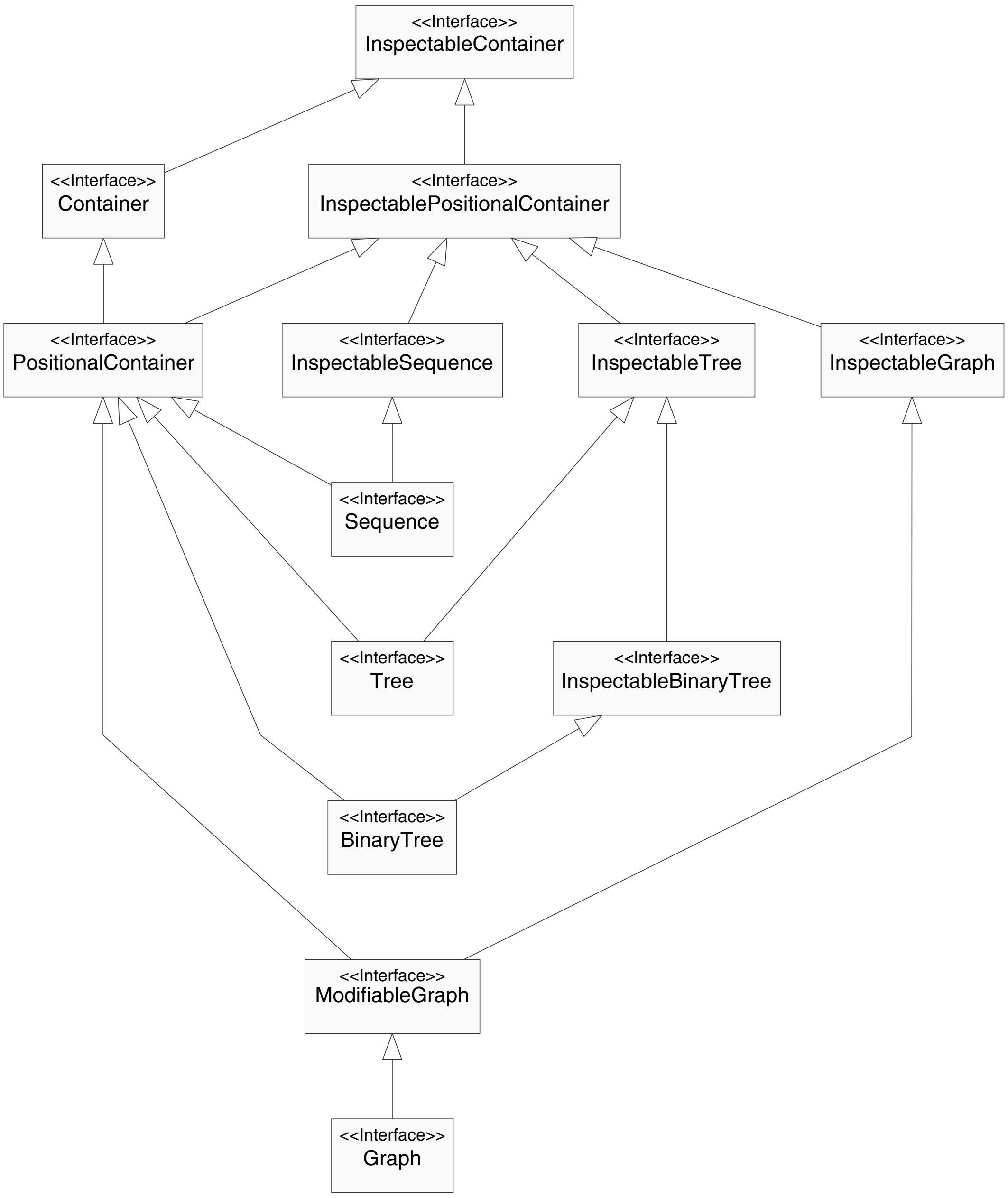


FIGURE 43.4: The positional containers interface hierarchy.

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 positions (positions()) or the elements (elements()) of the container.

Sequences

A sequence is a basic data structure used for storing elements in a linear, ranked fashion (see Chapter 2). Sequences can be implemented in many ways, e.g., as a linked list of nodes or on top of an array. In JDSL, sequences are described by interfaces InspectableSequence and Sequence, which extend InspectablePositionalContainer and PositionalContainer, respectively. In addition to the basic methods common to all positional containers, the sequence interfaces provide methods to access and modify positions at the sequence ends (methods such as first(), insertLast() and removeFirst()) or specific positions along the sequence (methods such as after(Position), atRank(int), insertBefore(Position) and removeAtRank(int)).

NodeSequence is an implementation of Sequence based on a doubly linked list of nodes. The nodes are the positions of the sequence. It takes $O(1)$ time to insert, remove, or access both ends of the sequence or a position before or after a given one, while it takes $O(n)$ time to insert, remove, or access positions at a given rank in the sequence. Thus, NodeSequence instances can be suitably used as stacks, queues, or deques.

ArraySequence is an implementation of Sequence based on a growable array of positions. Instances can be created with an initial capacity, and can be told whether or not to reduce this capacity when their size drops below a certain value, depending on whether the user prefers space or time efficiency. It takes $O(1)$ time to access any position in the sequence, $O(1)$ amortized time over a series of operations to insert or remove elements at the end of the sequence, and $O(n)$ time to insert or remove elements at the beginning or middle of the sequence. Hence, ArraySequence instances can be suitably used for quick access to the elements after their initial insertion, when filled only at the end, or as stacks.

Trees

Trees allow more sophisticated relationships between elements than is possible with a sequence: they allow relationships between a child and its parent, or between siblings of a parent (see Chapter 3). InspectableTree and Tree are the interfaces describing a general tree; they extend InspectablePositionalContainer and PositionalContainer, respectively. InspectableBinaryTree, which extends InspectableTree, and BinaryTree, which extends PositionalContainer, are the interfaces describing a binary tree. In addition to the basic methods common to all positional containers, the tree interfaces provide methods to determine where in the tree a position lies (methods such as isRoot(Position) and isExternal(Position)), to return the parent (parent(Position)), siblings (siblings(Position)) or children (methods such as children(Position), childAtRank(Position,int) and leftChild(Position)) of a position, and to cut (cut(Position)) or link (link(Position,Tree)) a subtree.

NodeTree is an implementation of Tree based on a linked structure of nodes. The nodes are the positions of the tree. It is the implementation to use when a generic tree is needed or for building more specialized (nonbinary) trees. NodeTree instances always contain at least one node.

NodeBinaryTree is an implementation of BinaryTree based on a linked structure of nodes. The nodes are the positions of the tree. Similarly to NodeTree, NodeBinaryTree instances always contain at least one node; in addition, each node can have either zero or two children. If a more complex tree is not necessary, using NodeBinaryTree instances will be faster and easier than using NodeTree ones.

Graphs

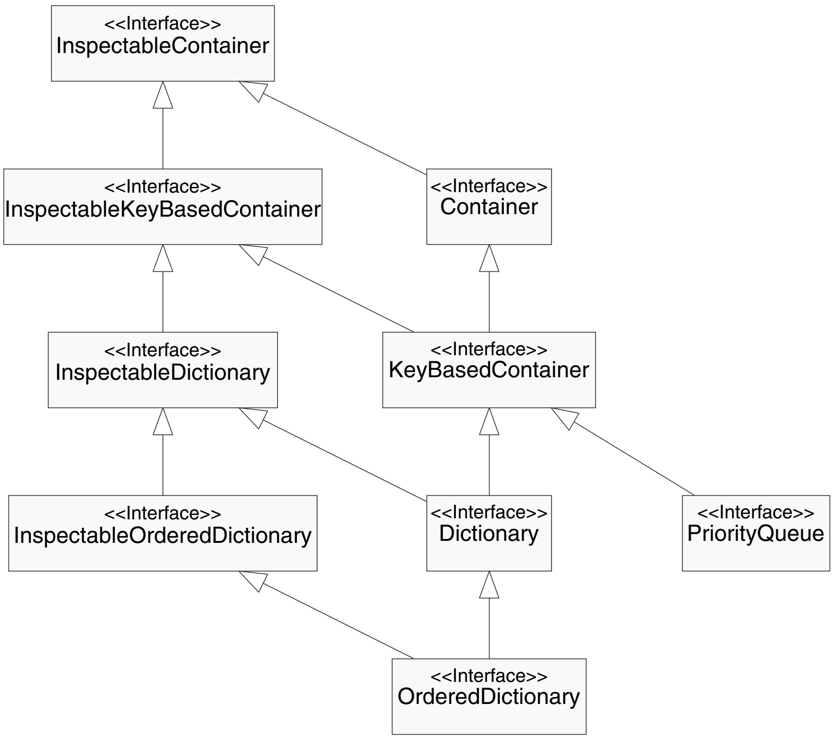
A graph is a fundamental data structure describing a binary relationship on a set of elements (see Chapter 4) and it is used in a variety of application areas. Each vertex of the graph may be linked to other vertices through edges. Edges can be either one-way, *directed* edges, or two-way, *undirected* 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.

The main graph interfaces are InspectableGraph, which extends InspectablePositionalContainer, ModifiableGraph, which extends PositionalContainer, and Graph, which extends both InspectableGraph and ModifiableGraph. These interfaces provide methods to determine whether two vertices are adjacent (areAdjacent(Vertex,Vertex)) or whether a vertex and an edge are incident (areIncident(Vertex,Edge)), to determine the degree of a vertex (degree(Vertex)), to determine the origin (origin(Edge)) or destination (destination(Edge)) of an edge, to insert (insertVertex(Object)) or remove (removeVertex(Vertex)) a vertex, to set the direction of an edge (setDirectionFrom(Edge,Vertex) and setDirectionTo(Edge,Vertex)), to insert (insertEdge(Vertex,Vertex,Object)), remove (removeEdge(Edge)), split (splitEdge(Edge, Object)), or unsplit (unsplitEdge(Vertex,Object)) an edge.

IncidenceListGraph is an implementation of Graph. As its name suggests, it is based on an incidence list representation of a graph.

43.3.3 Key-Based Containers

All key-based containers implement interfaces InspectableKeyBasedContainer and KeyBasedContainer, which extend InspectableContainer and Container, respectively (see Figure 43.5). 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



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

FIGURE 43.5: The key-based containers interface hierarchy.

(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 over the locators (locators()), the keys (keys()) or the elements (elements()) of the container.

Priority queues

A priority queue is a data structure used for storing a collection of elements prioritized by keys, where the smallest key value indicates the highest priority (see Part II). 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.

Interface PriorityQueue extends KeyBasedContainer. 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 the priority of an element can be changed using method replaceKey(Locator,Object), inherited from KeyBasedContainer.

ArrayHeap is an efficient implementation of PriorityQueue based on a heap. Inserting, removing, or changing the key of a key/element pair takes $O(\log n)$ time, while examining the key/element pair with the minimum key takes $O(1)$ 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.

Dictionaries

A dictionary is a data structure used to store key/element pairs and then quickly search for them using their keys (see Part III). An ordered dictionary is a particular dictionary where a total order on the set of keys is defined. All JDSL dictionaries are *multi-maps*, i.e., they can store multiple key/element pairs with the same key.

The main 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 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 is an implementation of Dictionary. As its name suggests, it is based on a hash table. Insertions and removals of key/element pairs 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 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.

43.3.4 Algorithms

In addition to the data structures described above, JDSL includes various algorithms that operate on them.

Sequence sorting

JDSL provides a suite of sorting algorithms for different applications. They all implement the SortObject interface, whose only method is sort(Sequence,Comparator). Sorting algorithms with the prefix List are most efficient when used on instances of NodeSequence while those with the prefix Array are most efficient when used on instances of ArraySequence.

ArrayQuickSort is an implementation of the quicksort algorithm. This algorithm runs in $O(n \log n)$ expected time and performs very well in practice. Its performance, however, degrades greatly if the sequence is already very close to being sorted. Also, it is not *stable*, i.e., 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 and ArrayMergeSort are two implementations of the mergesort algorithm. This algorithm is not as fast as quicksort in practice, even though its theoretical time complexity is $O(n \log n)$. There are no cases where its performance will degrade due to peculiarities in the input data, and it is a stable sort.

HeapSort is an implementation of the heapsort algorithm, and uses an instance of ArrayHeap (see Section 43.3.3) as a sorting device. Its performance, like that of mergesort, will not degrade due to peculiarities in the input data, but it is not a stable sort. Its theoretical time complexity is $O(n \log n)$.

Iterator-based tree traversals

JDSL provides two types of tree traversals. The first type is based on iterators: the tree is passed as an argument to the iterator constructor and is then iterated over using methods hasNext() and nextPosition(). Iterators give a quick traversal of the tree in a specific order, and are the proper traversals to use when this is all that is required. We recall that iterators in JDSL have snapshots semantics (see Section 43.2.2).

A preorder iterator visits the nodes of the tree in preorder, i.e., it returns a node before returning any of its children. Preorder iterators work for both binary and general trees; they are implemented in class PreOrderIterator.

A postorder iterator visits the nodes of the tree in postorder, i.e., it returns a node after returning all of its children. Postorder iterators work for both binary and general trees; they are implemented in class PostOrderIterator.

An inorder iterator visits the nodes of the tree in inorder, i.e., it returns a node in between its left and right children. Inorder iterators work only for binary trees; they are implemented in class InOrderIterator.

Euler tour tree traversal

The second type of tree traversals in JDSL is named Euler tour: it is implemented — in class EulerTour — as an algorithm object, which can be extended by applying the template method pattern.

The Euler tour visits each node of the tree several times, namely, a first time before traversing any of the subtrees of the node, then 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) and visitLastTime(Position), if the node is internal, or method visitExternal(Position), if the node is a leaf, is automatically called. A particular computation on the visited tree may be performed by suitably overriding those methods in a subclass of EulerTour.

The Euler tour is more powerful than the iterators described above as it can be used to implement more general kinds of algorithms. Note that, unlike the iterators, the Euler tour does not have snapshot semantics; this means that any modification of the tree during the execution of the Euler tour will cause undefined behavior.

Graph traversals

The depth-first search (DFS) traversal of a graph is available in JDSL. Depth-first search proceeds by visiting an unvisited vertex adjacent to the current one; if no such vertex exists, then the algorithm backtracks to the previous visited vertex.

Similarly to the Euler tour, depth-first search is implemented in JDSL as an algorithm object, which can be extended by applying the template method pattern. The basic implementation of depth-first search — DFS — is designed to work on undirected graphs. The user can specify actions to occur when a vertex is first or last visited or when different sorts of edges (such as "tree" edges of the DFS tree or "back" edges to previously visited vertices) are traversed by subclassing DFS and suitably overriding some methods.

DFS has two subclasses: FindCycleDFS, an algorithm for determining cycles in an undirected graph, and DirectedDFS, a depth-first search specialized for directed graphs. In turn, DirectedDFS has one subclass: DirectedFindCycleDFS, an algorithm for determining cycles in a directed graph. These subclasses are examples of how to apply the template method pattern to DFS in order to implement a more specific algorithm.

Topological numbering

A topological numbering is a numbering of the vertices of a directed acyclic 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, which decorates each vertex with a unique number, and UnitWeightedTopologicalNumbering, which decorates each vertex with a nonnecessarily unique number based on how far the vertex is from the source of the graph. Both topological numbering algorithms extend abstract class AbstractTopologicalSort.

Dijkstra's algorithm

Dijkstra's algorithm computes the shortest path from a specific vertex to every other vertex of a weighted connected graph. The JDSL implementation of Dijkstra's algorithm — IntegerDijkstraTemplate — follows the template method pattern and can be easily extended to change its functionality. Extending it makes it possible, for instance, to set the function for calculating the weight of an edge, to change the way the results are stored, or to stop the execution of the algorithm after computing the shortest path to a specific vertex (as done in subclass IntegerDijkstraPathfinder). An example of using Dijkstra's algorithm is given in Section 43.4.

The Prim-Jarník algorithm

The Prim-Jarník algorithm computes a *minimum spanning tree* of a weighted connected graph, i.e., a tree that contains all the vertices of the graph and has the minimum total weight over all such trees. The JDSL implementation of the Prim-Jarník algorithm — IntegerPrimTemplate — follows the template method pattern and can be easily extended to change its functionality. Extending it makes it possible, for instance, to set the function for calculating the weight of an edge, to change the way the results are stored, or to stop the execution of the algorithm after computing the minimum spanning tree for a limited set of vertices.

43.4 A Sample Application

In this section we explore the implementation of a sample application using JDSL. In particular, we show the use of some of the concepts described above, such as the graph and priority queue data structures, locators, decorations, and the template method pattern.

43.4.1 Minimum-Time Flight Itineraries

We consider the problem of calculating a minimum-time flight itinerary between two airports. The flight network can be modeled using a weighted directed graph: each vertex of the graph represents an airport, each directed edge represents a flight from the origin airport to the destination airport, and the weight of each directed edge is the duration of the flight. The problem we are considering can be solved by computing a shortest path between two vertices of the directed graph or determining that a path does not exists. To this purpose, we can suitably modify the classical algorithm by Dijkstra [2], 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 thus far. The priority queue is initialized by inserting vertex s with key 0 and all the other vertices with key $+\infty$ (some very large number). The algorithm repeatedly executes the following two 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 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).

43.4.2 Class IntegerDijkstraTemplate

As seen in Section 43.3.4, JDSL provides an implementation of Dijkstra's algorithm that follows the template method pattern. The abstract class implementing Dijkstra's algorithm is jdsl.graph.algo.IntegerDijkstraTemplate (see Figures 43.6–43.8; for brevity, the Javadoc comments present in the library code have been removed). The simplest way to run the algorithm is by calling execute(InspectableGraph,Vertex), which first initializes the various auxiliary data structures with init(g,source) and then repeatedly calls doOneIteration(). Note that the number of times doOneIteration() is called is controlled by shouldContinue(). Another possibility, instead of calling execute(InspectableGraph,Vertex), is to call init(InspectableGraph,Vertex) 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

```
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(0);
  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);
  }
```

FIGURE 43.6: Class IntegerDijkstraTemplate.

```
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 g_.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);
  }
}
```

FIGURE 43.7: Class IntegerDijkstraTemplate (continued).

```
  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 < 0 || 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
```

FIGURE 43.8: Class IntegerDijkstraTemplate (continued).

in the priority queue in constant time, whenever its key has to be modified. This is possible through the locator accessors provided by class jdsl.core.ref.ArrayHeap (see Section 43.3.3). In init(InspectableGraph,Vertex), each vertex u of the graph is inserted in the priority queue and a locator uLoc for the key/element pair is returned. By calling setLocator(u,uLoc), each vertex u is decorated with its locator uLoc; variable LOCATOR is used as the attribute name. Later, in doOneIteration(), the locator of vertex v is retrieved by calling getLocator(v) in order to access and possibly modify the key of v; we recall that the key of v is the shortest known distance from the source vertex 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 by calling 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. Following the template method pattern, they call some methods, namely, shouldContinue(), vertexNotReachable(Vertex), shortestPathFound(Vertex,int), and edgeRelaxed(Vertex,int,Edge, int,Vertex,int), that may be overridden in a subclass 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_; variable DISTANCE 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. Other important methods are isFinished(Vertex), which returns whether a given vertex is marked as finished, and distance(Vertex), which returns the shortest distance from source_ to a given finished vertex.

43.4.3 Class IntegerDijkstraPathfinder

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 in abstract class jdsl.graph.algo.IntegerDijkstraPathfinder (see Figure 43.9; for brevity, the Javadoc comments present in the library code have been removed), which extends IntegerDijkstraTemplate. The algorithm is run by calling execute(InspectableGraph,Vertex,Vertex). 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 (pathExists()), and, in this case, to return it (reportPath()).

43.4.4 Class FlightDijkstra

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 from the beginning of the travel to the arrival at the airport represented by that vertex. In Figure 43.10 we show the code of class FlightDijkstra; this class is part of the tutorial** distributed with JDSL. All it takes to implement our

**`http://www.jdsl.org/tutorial/tutorial.html`

```
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 IntegerDijkstraPathfinder
```

FIGURE 43.9: Class IntegerDijkstraPathfinder.

application is to override method incidentEdges(), so that only the outgoing edges of a finished vertex are examined, and to define method weight(Edge). As noted before, the weighted graph representing the flight network is a directed graph. Each edge stores, as an element, an instance of auxiliary class FlightSpecs providing the departure time and the duration of the corresponding flight. Note that the weights of the edges are not determined before the execution of the algorithm, but rather depend on the computed shortest distance

```
import jdsl.graph.api.*;
import jdsl.graph.algo.IntegerDijkstraPathfinder;
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);
  }

}
```

FIGURE 43.10: Class FlightDijkstra.

between the source vertex and the origin of each edge. Namely, they are obtained by adding the duration of the flight corresponding to the edge and the connecting time at the origin airport for that flight.[††] Method TimeTable.diff(int,int) simply computes the difference between its two arguments modulo 24 hours. The algorithm is run by calling execute(InspectableGraph,Vertex,Vertex,int), where the fourth argument is the earliest time the passenger can begin traveling.

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

Acknowledgments

We would like to thank all the members of the JDSL Team for their fundamental role in the design, implementation, and testing of JDSL. It has been a great experience and a real pleasure working together.

[††]In this sample application we ignore the minimum connecting time requirement, which however could be accommodated with minor code modifications.

This work was supported in part by the National Science Foundation under grants CCR-0098068 and DUE-0231202.

References

[1] 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.

[2] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[3] 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.

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

[5] 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.

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

[7] M. T. Goodrich and R. Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples.* John Wiley & Sons, New York, NY, 2002.

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

[9] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison-Wesley, Reading, MA, 2nd edition, 2001.

[10] R. Tamassia, M. T. Goodrich, L. Vismara, M. Handy, G. Shubina, R. Cohen, B. Hudson, R. S. Baker, N. Gelfand, and U. Brandes. JDSL: The data structures library in Java. *Dr. Dobb's Journal*, (323):21–31, Apr. 2001.