A Case Study in Algorithm Engineering for Geometric Computing

Roberto Tamassia and Luca Vismara

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-97-18
December 1997

A Case Study in Algorithm Engineering for Geometric Computing*

Roberto Tamassia *Luca Vismara*
Center for Geometric Computing
Department of Computer Science
Brown University
Providence, Rhode Island 02912–1910
{rt,lv}@cs.brown.edu

Abstract

The goal of this paper is to prove the applicability of some advanced software design concepts to geometric computing through a vertical case study. The work is presented within the framework of the GeomLib project, aimed at developing an easy to use, reliable, open library of robust and efficient geometric algorithms. We present the criteria that have inspired the preliminary design of GeomLib and discuss the guidelines that we have followed in the initial implementation.

*Research supported in part by the U.S. Army Research Office under grant DAAH04-96-1-0013 and by the National Science Foundation under grant CCR–9423847. Portions of the results described in this paper have been presented at the *Workshop on Geometric Computing*, Sophia Antipolis, France, 1997, and at the *Workshop on Algorithm Engineering (WAE '97)*, Venezia, Italy, 1997.

1 Introduction

The GeomLib project addresses the important objective of developing an easy to use, reliable, open library of robust and efficient geometric algorithms. Recent object-oriented design concepts such as design patterns [24] and algorithm abstraction [63] are extensively used throughout the entire project. In the design phase, we have taken into account the experience of other similar efforts, such as the Library of Efficient Data structures and Algorithms (LEDA) [7, 43, 44, 46] and the more recent Computational Geometry Algorithms Library (CGAL) [20, 50, 57, 65].

Two are the major goals of the project:

- To provide researchers in computational geometry with a framework for algorithm engineering, with a specific emphasis on geometric computing. In this context, GeomLib will be typically used for rapid prototyping and experimental studies of geometric algorithms.
- To make computational geometry results available to the users in other areas, such as robotics, geographic information systems, mechanical engineering, computer graphics, etc.

It has been noted [10, 42, 62] that to address applications from those areas is a strategic direction for computational geometry. In particular, the importance of the above goals stems from the following observations:

- Programming effort is often unnecessarily expended reimplementing fundamental geometric algorithms.
- Innovative geometric algorithms are often not implemented, because their creators do not have the interest, the skills, or the resources to accompany the algorithm with an implementation.
- In the implementation of geometric algorithms, important aspects, such as robustness requirements or degeneracy conditions, are often overlooked, with detrimental effects on the correctness of the results.

We present the criteria that have inspired the preliminary design of GeomLib and discuss the guidelines that we have followed in the initial implementation. We exemplify the innovative aspects of our design through the discussion of a vertical case study: the design and implementation of a new algorithmic pattern, called binary space partition search, that provides a unifying description of two well-known algorithms for the fundamental geometric problem of planar point location.

GeomLib is part of a larger project, named JDSL [25, 27], aimed at constructing an innovative library of data structures and algorithms using the Java programming language. In designing those parts of JDSL necessary for the case study, we have defined a collection of interfaces that describe some fundamental data structures used in geometric computing, suitably arranged in hierarchies. For each interface, different implementations are or will be provided. This will allow the users to choose the most appropriate one considering their efficiency constraints. Users of JDSL ideally will develop algorithms using interfaces rather than specific implementations, thus creating more general code.

In the design of GeomLib, the combinatorial, topological, and geometric properties of geometric objects are accessed through different, increasingly specialized interfaces. In the implementation of the binary space partition search algorithmic pattern, for example, the geometric information of the planar subdivision is accessed only through few, clearly identified interface methods; most of the algorithmic pattern is actually implemented accessing only the topological information of the

planar subdivision. As a result, a change in the particular arithmetic representation chosen for the basic geometric objects (points and linear curves) that compose the planar subdivision does not require any modifications of the algorithmic pattern implementation itself.

GeomLib, as any library, may suffer from a relative inefficiency: providing generality usually requires an overhead with respect to an ad hoc program to solve a certain problem. As described before, however, one of the main purposes of GeomLib is to provide a framework for rapid prototyping of algorithms, which can then be reimplemented as stand-alone applications, if necessary. Another possible source of inefficiency arises from the choice of Java as the implementation language. Its cross-platform capability comes at the price of a reduced execution speed. However, in the near future the difference in execution speed between a Java program and a, say, C++ program should become insignificant thanks to the use of second-generation Java virtual machines or high performance compilers that produce optimized platform-specific native code.

The rest of the paper is organized as follows. In Section 2 a set of requirements for a geometric computing library is described. Previous work is discussed in Section 3, where the following approaches are evaluated with respect to the requirements: non-integrated collections of algorithms, such as "graphics gems" and "numerical recipes", specialized libraries for scientific computing, and completely integrated libraries, such as LEDA and CGAL. The preliminary design of JDSL/GeomLib is presented in Section 4 through a description of the main components of the architecture. In Section 5, we analyze our design with respect to the previously described requirements. The vertical case study is presented in Section 6. Our plans for the future development of GeomLib are summarized in Section 7. Finally, in Appendix A we recall some object-oriented concepts that are used throughout the paper, while the choice of Java as the implementation language for the current JDSL/GeomLib prototype is motivated in Appendix B.

2 Requirements for a Geometric Computing Library

In this section, we describe a set of requirements that a library for geometric computing should satisfy. Many of these requirements apply to any software library, while some others are particularly important for geometric computing.

Ease of Use The library should be easy to use. Names of components and operations should be intuitive and standardized. Documentation should be integrated in the library. Knuth pioneered integration between code and documentation with the concept of "literate programming" (see, e.g., [37]). Note that the Java programming language provides this capability as part of its specification [1].

Efficiency The methodologies and techniques used in the design of the library (abstraction, generality, object-orientation) should introduce a low *overhead* with respect to an ad hoc program to solve a certain problem. Efficiency of algorithms and data structures should not be evaluated only through the standard asymptotic analysis measures; constant factors should also be considered, together with new efficiency measures for geometric algorithms, such as the *degree* [41].

Reliability Although geometric algorithms are easier to express in the real-RAM model, issues such as *robustness* or the use of *external memory* should be addressed by a practical geometric library. For example, exact rational arithmetic and support for data structures that efficiently work with secondary storage should be provided. Another important reliability criterion is

the detailed handling of all inputs, including *degenerate* ones [20, 62]. Note, however, that a trade-off exists between reliability and other criteria, in particular efficiency.

Openness Multiple implementations should be provided for each data structure. Being able to choose among multiple implementations is a very powerful capability; it gives the possibility to experiment in order to choose the most appropriate one for a given problem. Of particular importance, in geometric computing, is the choice of the data types. Possible examples are the arithmetic representation for geometric objects, or the use of multilevel spatial data descriptions, providing "zooming" capabilities, in geographical information systems (see, e.g., [4, 64]).

Extensibility The architecture of a library should be extensible by the decentralized contributions of the community of its users, while maintaining evolving standards that enable the contributions to interoperate. Extensible architectures should be contrasted with *monolithic* ones. The great success of various projects, e.g., the LAPACK library, the GNU software system, the Unix operating systems, and the Internet (with its Request For Comments documents), testify the importance of extensibility, especially in the presence of standards for collaboration.

Reusability Reuse of design and implementation should be maximized, both internally and externally. By *internal reusability* we mean that the library uses its own components and programs to build more complex ones. By *external reusability* we mean that components of external libraries should also be used, wherever feasible. Resources are always limited, and many of the existing libraries are very good. Portability of the implementation language(s) and extensibility are essential to achieve a high degree of external reuse.

Modularity Some of the most relevant concepts of structured programming are modularity and layered design, which were introduced in languages like Ada. Large software applications are divided in loosely coupled *modules*, usually arranged in layers. More recently the concept of *component* has been introduced. Components are not arranged in a strictly hierarchical way, since not always a true hierarchical relationship exists among them. They allow greater flexibility, avoiding artificial and unnecessary dependencies. Modularity increases reusability and decreases the cost of reliability.

Functionality The library should provide a significant subset of the existing geometric computing algorithms. Reusability of existing work and extensibility are crucial for achieving functionality.

Correctness Checking It is a well-known fact that programming is an error-prone task. On the other hand, *program testing* cannot guarantee the correctness of a program, and *formal proof* of correctness do not seem to be applicable in practice. As a possible solution to this impasse, the concept of *program checking* has been recently introduced (see, e.g., [5, 6, 11, 21, 45]. Rather than proving the correctness of the program for any possible input, each time the program is run (with a specific input), the correctness of its output (with respect to that input) is checked. A program checker should satisfy certain requirements: it should be *correct* itself, *simple*, so that its correctness can be established beyond any reasonable doubt, and *efficient*, i.e., requiring less resources than the checked program. Program checkers should be integrated within the library to boost confidence in the algorithm implementations. However, as noted in [45], a trade-off exists between correctness checking and other criteria, such as efficiency and extensibility. Checkers may also provide valuable insight into the corresponding algorithms, when combined with animation and visualization techniques.

3 Previous Work

Previous related work on algorithm libraries includes the gems/recipes approach, specialized libraries, and the European initiatives LEDA and CGAL. We evaluate previous work according to the requirements presented in Section 2.

3.1 Gems and recipes

"Graphics gems" [2, 26, 31, 35, 51] and "numerical recipes" [55] have proved very useful in computer graphics and scientific computing, respectively. Through books and available source code, the gems/recipes approach has disseminated efficient and reliable procedures for computer graphics and scientific computing to a wide audience.

There are two main objections to this approach as a model for a geometric computing library. Numerical recipes do not use complex data structures; typically the most complex ones are dense arrays. In contrast, geometric computing requires substantially more complex data structures. And because of the relative lack of complexity, users of gems and recipes tend to improve the efficiency of their applications by directly implementing them without any substantial abstraction. This, however, is possible only because of the relative simplicity of the gem or recipe to be implemented, as well as the relative homogeneity of the problem elements. It does not seem applicable to the conceptually more difficult geometric problems.

The Directory of Computational Geometry Software created by Nina Amenta and available at `http://www.geom.umn.edu/software/cglist/` is a comprehensive collection of "gems" in the area of geometric computing. An example of excellent program from that collection is the widely used Qhull, by Barber, Dobkin, and Huhdanpaa.

3.2 Specialized Libraries

A wide variety of specialized libraries exist for scientific computing, operations research, and other domains. Each library is specialized in one particular type of problems, e.g., linear algebra, linear programming, fluid dynamics, stress analysis, or molecular biology. Specialized libraries are widely used, invaluable, and often quite expensive. They provide functionality for their specific domain; this implies that they are often very efficient and reliable with respect to that domain. They are generally easy to use, as they match the expectations of the users. Proprietary libraries, by their nature, do not provide extensibility, but a number of specialized libraries are collaboratively developed and are extensible to some extent. See, e.g., BLAS and LAPACK available from the Netlib repository at `http://www.netlib.org`. Almost all large libraries provide some degree of correctness checking through a validation suite.

These libraries, however, are heavily specialized. Reuse is generally limited when they are used outside of their domain because of interoperation difficulties. (Interoperation requires a spectrum of standards, ranging from the conceptual level to the implementation level.) Many have limited data type support, being written in Fortran, and thus lack modularity.

Some small specialized libraries have been written for computational geometry, usually accompanying introductory textbooks (see e.g., [38, 49]). They are typically easy to use, but lack functionality.

3.3 LEDA

LEDA [44, 46], the Library of Efficient Data Structures and Algorithms, was not designed with the exclusive goal of supporting geometric computing. However, this is one of its principal uses and the focus of much of the continuing research activity [7, 43]. It is available at `http://www.mpi-sb.mpg.de/LEDA`.

Ease of Use LEDA's data structures and algorithms are well documented with respect to their space and time complexity. Every new release is accompanied by a detailed user manual.

Efficiency LEDA provides implementations of many asymptotically optimal algorithms. Templates[1] are extensively used in order to make the algorithms generic yet time-efficient.

Reliability Multiple types of arithmetic, such as arbitrary length integer numbers, rational numbers, or the use of floating point filters allow the definition of an exact homogeneous coordinate system. Different types of arithmetic cannot currently coexist in geometric data structures, although this might be possible in the future exploiting name spaces (a new capability of C++).

Openness Multiple implementations are provided for many data structures, with the appropriate implementation being selected through the template mechanism. For example, there exist implementations of a priority queue as a Fibonacci heap, a pairing heap, a k-nary heap, a monotonic heap, a van Emde Boas tree. It is even possible to add a user-written implementation, as long as it conforms to the standard interface of a priority queue. Templates are also used to parameterize the information type(s) of a collection data type. This degree of flexibility is less available for the far more complex data structures used in computational geometry, because of the inherent limitations of the template mechanism when dealing with complex modeling relationships. Points in the geometric kernel are represented as rational numbers using homogeneous coordinates; it is not possible to choose a different representation for the points, if desired. Also, for each geometric test, the computation performed for its evaluation is fixed. As an example, we consider testing the vertical position of a point with respect to a line. This test usually requires the computation of the sign of a determinant. However, as discussed in Section 6.2, there are cases in which this predicate could be evaluated in a more efficient way.

Extensibility LEDA has proven to be extensible from the user community. A large number of specialized packages written by users for tasks like map labeling, visibility algorithms, and location problems are available.

Reusability LEDA practices internal reuse extensively. Basic data structures (sequences, priority queues, dictionaries, and union-find structures) are reused, as components of more sophisticated ones. External reuse is more limited, perhaps because of portability concerns.

Modularity LEDA is evolving from an initial flat design to a layered one, as exemplified by the exact, higher-dimensional geometric kernel recently incorporated [43]. The geometric kernel is composed by an arithmetic layer, a linear algebra layer, and a geometric layer.

[1] Templates are a code generation mechanism of the C++ programming language that allows the definition of functions and classes to be parameterized with respect to one or more data types. For example, a set templated class can be defined with the notation `set <class T>`, and later specialized to a set of integers class with the notation `set <int>`.

Functionality LEDA implements an impressive collection of modern data structures and algorithms.

Correctness Checking Pioneering work on correctness checkers has been done within LEDA [21, 45].

3.4 CGAL

CGAL and LEDA are separate but complementary initiatives. CGAL [20, 50, 57, 65], the Computational Geometry Algorithms Library, is a recently started initiative involving seven research centers and funded by the European Community. CGAL is still work in progress, but a first version has been recently released. It consists of three parts: the kernel, which contains primitive geometric objects, the basic library, which contains a basic geometric data structures and algorithms; and the support library, which contains non-geometric support facilities. It is available at `http://www.cs.ruu.nl/CGAL`.

Ease of Use A detailed on-line documentation is available at the CGAL web site. The naming conventions have been designed very carefully. The support library provides, among others, I/O support for debugging and for interfacing CGAL to various visualization tools.

Efficiency As in LEDA, implementations of various asymptotically optimal geometric algorithms are provided. The user is allowed to specify the type of arithmetic used for representing primitive geometric objects, so that computationally expensive exact rational arithmetic is used only where actually needed. This is obtained through the C++ template mechanism.

Reliability CGAL provides exact rational arithmetic in the kernel, in order to have robust implementations of geometric algorithms.

Openness As mentioned above, the user is allowed to specify the type of arithmetic used for representing primitive geometric objects. Like in LEDA, however, the computation performed for evaluating a geometric test is fixed.

Extensibility A certain degree of extensibility is present, in particular for the possibility of importing user-designed arithmetic types (e.g., LEDA's `integer` and `real` types, or the GNU Multiple Precision Arithmetic Library).

Modularity CGAL does not hierarchically organize geometric objects, but has a layered design comparable to LEDA's. It supports interoperation between geometric objects through the container mechanism of the Standard Template Library [47].

Functionality The kernel contains a large collection of 2D and 3D primitive geometric objects and operations on them. Both Cartesian and homogeneous coordinate systems are provided. Implementations of various basic geometric data structures and algorithms are contained in the basic library.

Correctness Checking A sophisticated system of checks and warnings has been implemented for the kernel methods and the basic library algorithms.

4 The Preliminary Design of JDSL/GeomLib

The goal of this paper is not to describe the design of a comprehensive geometric computing library, but rather to prove the applicability of some advanced software design concepts to geometric

computing through a vertical case study. Thus, in this section, we will review only those aspects of JDSL/GeomLib that are relevant to the case study. The JDSL project is described in greater detail in [25, 27]. Our design relies on various object-oriented design concepts, such as *object*, *class*, *interface*, *inheritance*, *polymorphism*, *dynamic binding* and *type checking*, and *design pattern*. We briefly review them in Appendix A, and refer the interested reader to, e.g., [9, 24, 60, 66].

Of particular importance in the design of JDSL/GeomLib is the concept of interface. Ideally, users of JDSL/GeomLib use interfaces rather than classes as object types; actual classes need only be specified when creating an object. And if the abstract class factory design pattern [24] is implemented, also the creation of an object takes place through an interface. Interfaces are general, while classes are specialized: thus, using interfaces instead of specific classes creates more general code, allowing different classes, implementing the same interface, to be used interchangeably. In our design, we have exploited multiple interface inheritance and single class inheritance, thus taking advantage of the flexibility provided by the former and, at the same time, avoiding the well-known problems of multiple class inheritance.

A significative result of the analysis performed for the case study is the convenience of having multiple views of the same geometric object. In fact, a geometric object can be described at different levels of abstraction, considering its combinatorial, topological, and geometric properties separately. In our design, these properties are made accessible through different interfaces. We will use the geometric object *planar subdivision* as an example, and show how the different interfaces it implements correspond to different possible views.

The architecture of JDSL is partitioned into four different components, namely the *combinatorial*, the *topological*, the *geometric*, and the *arithmetic* components. In particular, GeomLib consists of the geometric and the arithmetic components. In our Java implementation, each component corresponds to one or more Java packages. Each package consists of a collection of interfaces, and for each interface a reference implementation is or will be provided. The interfaces are arranged in hierarchies that may extend across different packages or components. Note that the design of JDSL does not directly depend on Java; its validity extends to C++ and other object-oriented languages. The choice of Java as the implementation language for the current JDSL prototype is motivated in Appendix B.

In the rest of this section we will present a high-level view of the four components of JDSL. As said above, their full description is beyond the scope of this paper.

4.1 The Combinatorial Component

In this component, many fundamental data structures used in combinatorial algorithms are defined and implemented. A recent trend in the study of fundamental data structures is to present them under a general framework. In fact, most of them can be viewed as *containers* that store a collection of heterogeneous objects, called the *elements* of the container [25, 27, 47, 48].

This component is further divided into two subcomponents, one for the basic data structures and one for the combinatorial graph. A high-level view of the interface hierarchies for the combinatorial component is shown in Fig. 1.

4.1.1 The Basic Data Structures Subcomponent

In this subcomponent, most of the basic data structures are defined and implemented. Containers can be roughly divided into two categories: *positional* containers and *key-based* containers. Typical positional containers are *sequences*, *trees*, and *graphs*; typical key-based containers are *priority queues* and *dictionaries*. As the name suggests, positional containers are based on the concept

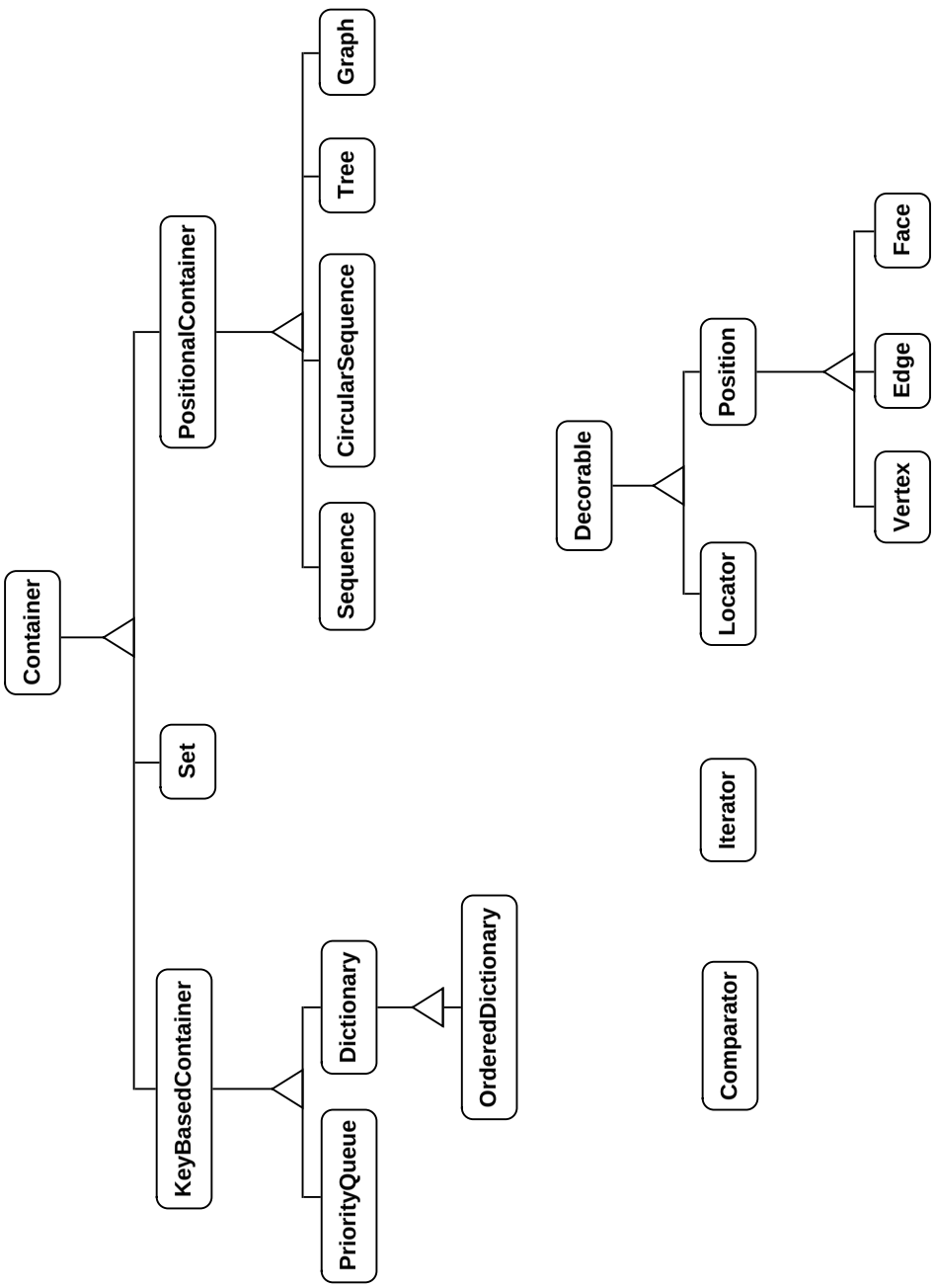

Figure 1: The container interface hierarchy of JDSL. Rounded rectangles represent interfaces.

of *position*. Positions abstract the common properties of the constituents of positional containers. Considering a sequence as an example, a position models both a node of a linked list implementation and an array entry of an array-based implementation. Each position stores an element of the container, and, given a position, it is possible to access its element and its container in constant time. Positions are fixed in a positional container, while elements can be moved from one position to another (as an example, we can imagine an element that is moved from the first to the last position of a sequence). On the other hand, the concept of position is not inherent in key-based containers, although each key-based container is, at the very end, implemented using a positional container (e.g., a binary tree used to implement a dictionary). A different mechanism is used to access a specific element in a key-based container in constant time; its description, however, is beyond the scope of this paper.

This subcomponent corresponds to the `jdsl.core` package of JDSL. Some of the most relevant interfaces for our case study are the following:

`Container` This interface describes a generic collection of heterogeneous objects and is the common root of the container hierarchy. It provides basic methods such as `isEmpty()`, `size()`, returning the number of elements in the container, and `elements()`, returning an enumeration of the elements in the container.

`Position` This interface describes the concept of position of a positional container (e.g., a node of a tree, or a vertex of a graph). The two main methods are `element()`, returning the element stored in the position, and `container()`, returning the container to which the position belongs.

`PositionalContainer` This interface extends the `Container` interface and is the common parent of the `Sequence`, `CircularSequence`, `Tree`, and `Graph` interfaces. It provides methods such as `positions()`, returning an enumeration of the positions in the container and `replace(Position,Object)`, replacing the element stored in the position with a new object.

`Decorable` This interface is used to implement the *decorator* design pattern [24]. The motivation of this pattern is to attach additional, named attributes to individual objects rather than to an entire interface or class. In our case the type of the objects we want to decorate is `Position`, which suitably extends `Decorable`. Typically, an attribute is a temporary piece of information, necessary for some specific computation (e.g., marking a vertex of a graph visited). An example is the decoration of the positions (nodes) of a tree with weight attributes. The interface provides methods such as `create(Object,Object)` and `destroy(Object)` for creating and removing an attribute, and `set(Object,Object)` and `get(Object)` for setting and getting the value of an attribute.

4.1.2 The Graph Subcomponent

In this subcomponent, the graph interface and some auxiliary interfaces are defined and implemented. The graph interface describes a graph as a *combinatorial* object, i.e., simply a set of elements and a binary relationship on this set. It inherits from the positional container interface in the basic data structures subcomponent and is further extended in the topological component. A high-level view of the graph hierarchy is shown in Fig. 2. Again, a detailed description of the graph hierarchy is beyond the scope of this paper.

This subcomponent corresponds to the `jdsl.graph` package of JDSL. The present interfaces are the following:

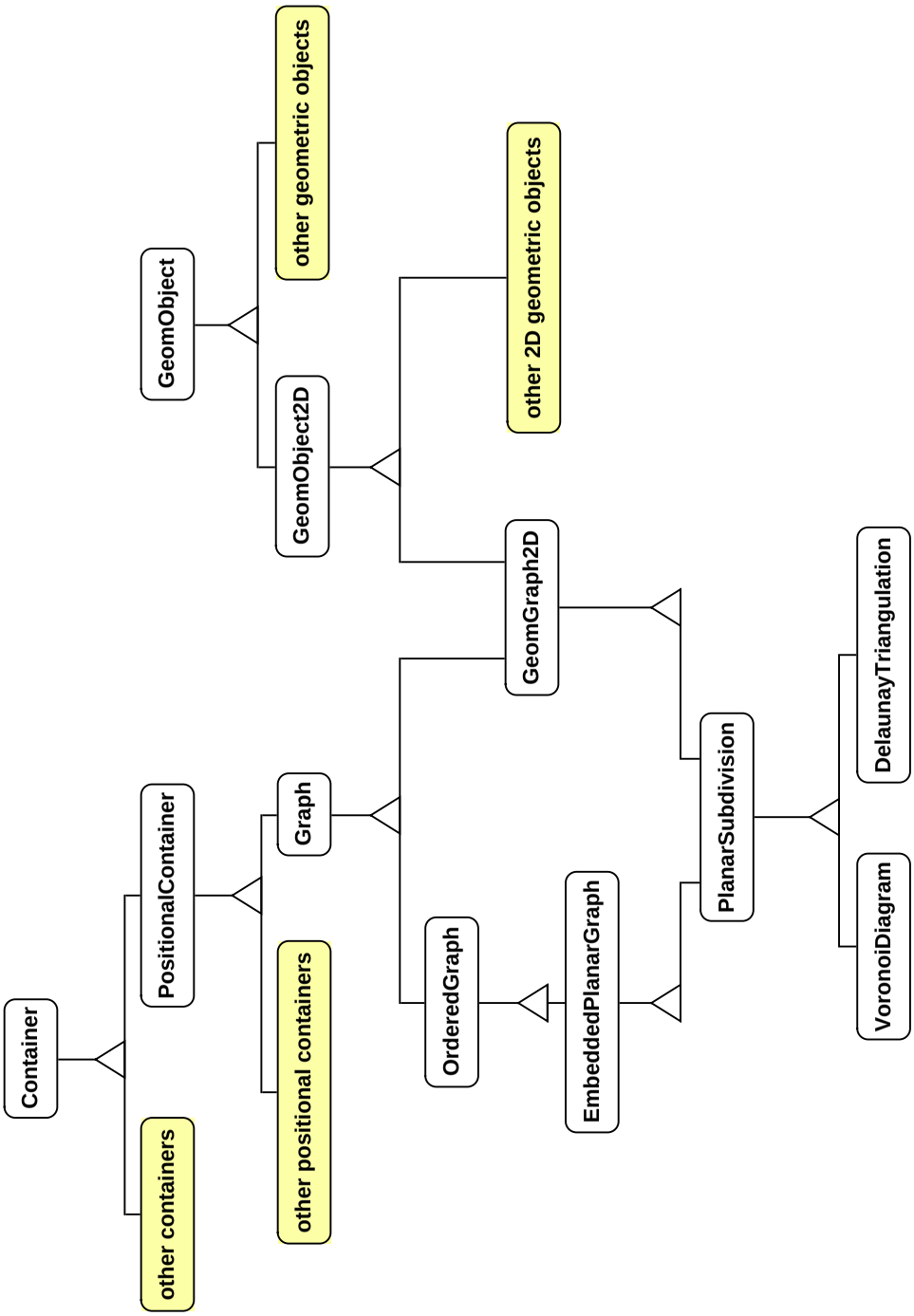

Figure 2: A high level view of the graph interface hierarchy of JDSL. Rounded rectangles represent interfaces.

`Vertex`, `Edge` These interfaces extend the `Position` interface. They provide no additional method but are used as "typing" interfaces, allowing a more effective type checking, e.g., when used as method parameters.

`Graph` This interface describes a combinatorial graph. It provides methods such as `numVertices()` and `numEdges()`, returning the number of vertices and edges of the graph, `adjacentVertices(Vertex)` and `incidentEdges(Vertex)`, for accessing the positions adjacent to or incident with a vertex, `origin(Edge)` and `destination(Edge)`, returning the origin and the destination of a directed edge, etc.

As an example of internal reuse of code, the existing implementation of the `Graph` interface uses an implementation of the `Sequence` interface to represent its adjacency lists. Note that since the `Sequence` implementation is used only through the interface methods, any `Sequence` implementation can be used, without having to change the `Graph` implementation.

4.2 The Topological Component

In this component, the ordered graph interface, the embedded planar graph interface and an auxiliary interface are defined and implemented. The ordered graph interface describes a graph as a *topological* object, i.e., a combinatorial graph with the additional information about the ordering of the edges around the vertices. Accordingly, the ordered graph interface inherits from the graph interface. An embedded planar graph is a planar ordered graph where the additional information about the ordering of the edges around the vertices is the one given by an embedding of the graph in the plane. Accordingly, the embedded planar graph interface inherits from the ordered graph interface. It is further extended in the geometric component.

This component corresponds to the `jdsl.map` package of JDSL. The present interfaces are the following:

`OrderedGraph` This interface describes a topological graph and extends `Graph`. Its main additional methods are `prevIncidentEdge(Vertex, Edge)` and `nextIncidentEdge(Vertex, Edge)`, returning the edge before or after an edge around a vertex.

`Face` This interface extends `Position`, and, like `Vertex` and `Edge`, is used as a "typing" interface.

`EmbeddedPlanarGraph` This interface extends `OrderedGraph`. It provides additional methods such as `numFaces()`, returning the number of faces of the graph, `incidentFaces(Vertex)`, returning the faces incident with a vertex, `leftFace(Edge)` and `rightFace(Edge)`, returning the face to the left or to the right of a directed edge, `dual()`, returning the topological dual embedded planar graph, etc.

There exist several possible representations for an embedded planar graph, e.g., the DCEL representation [54], the quad-edge representation [29], the dynamic representations described in [18, 19, 61], etc. Each representation presents advantages and disadvantages, and, typically, some may be more suitable than others for a specific application. For instance, in applications in which the embedded planar graph is frequently subject to insertions and deletions of vertices and edges, it is important to be able to efficiently update the representation. For some other applications it may be important to easily access the dual graph. Geographical information systems require representations specifically designed for efficient secondary storage access or for multilevel/multiresolution access (see, e.g., [4, 64]). The existing implementation of the `EmbeddedPlanarGraph` interface is based

on the *incidence graph* representation described in Chapter 11 of [15]. The embedded planar graph G is represented through a graph G' such that each vertex of G' corresponds to a vertex, edge, or face of G, and each vertex of G' corresponding to an edge e of G is adjacent to the four vertices corresponding to the endvertices and to the incident faces of e in G. The incidence graph representation has the nice property that it represents an embedded planar graph and its topological dual at the same time. As an example of internal reuse of code, the existing implementation of the `EmbeddedPlanarGraph` interface uses an implementation of the `OrderedGraph` interface to represent the incidence graph.

4.3 The Geometric Component

This component, together with the arithmetic component, form the GeomLib part of JDSL. It is further divided into two subcomponents, one for the basic geometric objects and one for the advanced geometric objects. The interface hierarchy for the geometric component is shown in Fig. 3.

4.3.1 The Basic Geometric Objects Subcomponent

In this subcomponent, basic geometric objects, such as *point*, *line*, *ray*, *segment*, *circle*, etc., are defined and implemented. Currently, interfaces have been defined and implementations provided only for two-dimensional basic geometric objects.

This subcomponent corresponds to the `jdsl.geomobj` package in the implementation of JDSL. Some of the most relevant interfaces are the following:

`GeomObject` This is the interface from which all the other geometric interfaces inherit. Its only two methods are `dim()`, returning the dimension of the geometric object, and `geomFactory()`, returning the geometric abstract factory that created the geometric object. It is extended by the "typing" interfaces `GeomObject2D` and `GeomObject3D`.

`GeomFactory2D` This interface implements the abstract factory design pattern [24] for the basic geometric objects. It provides methods such as `newPoint(int,int)`, `newSegment(Point2D,Point2D)`, `newCircle(Point2D,Point2D,Point2D)`, etc. Geometric abstract factories provide a common interface for creating families of related or dependent geometric objects without specifying their classes. This is obtained through a simple grouping of the constructors of the geometric objects as methods of the geometric abstract factory.

`Point2D` This interface describes a two-dimensional point. Its only two methods are `x()` and `y()`, returning a `double` approximation of the point coordinates.

`Curve2D`, `OpenCurve2D`, `ClosedCurve2D` These are "typing" interfaces.

`LinearCurve2D` This interface describes a rectilinear curve. It inherits from `OpenCurve2D` and is further extended by interfaces `Line2D`, `Ray2D`, and `Segment2D`. It provides methods such as `points()`, returning its two defining points, `isHorizontal()`, and `isVertical()`.

`GeomTester2D` This interface is a collection of various two-dimensional geometric tests. Some of the most relevant methods are `aboveBelow(LinearCurve2D,Point2D)`, testing whether the point is above, on, or below the rectilinear curve, `leftRight(LinearCurve2D,Point2D)`, testing whether the point is to the left, on, or to the right of the rectilinear curve, the similar methods `aboveBelow(Point2D,Point2D)` and `leftRight(Point2D,Point2D)` where the test

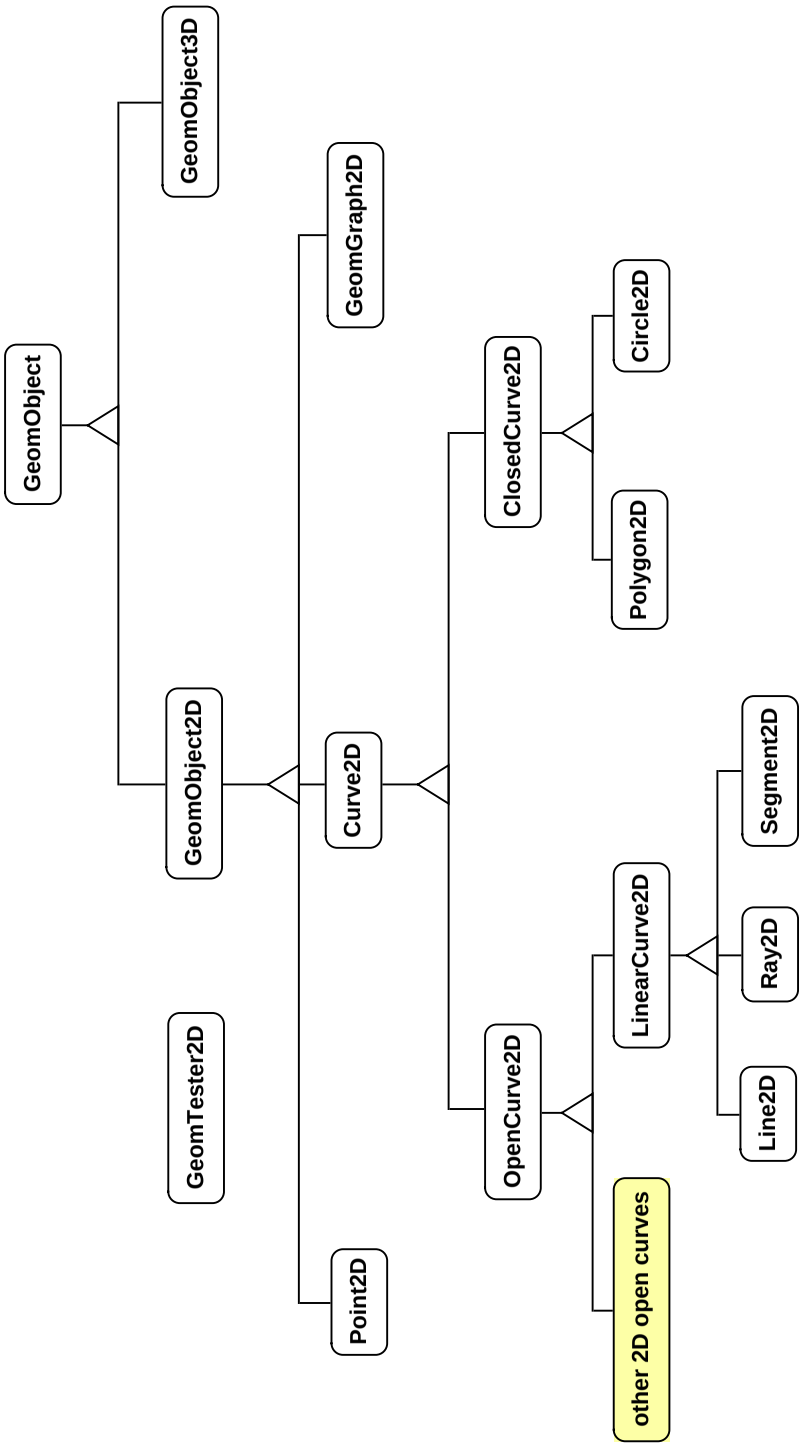

Figure 3: The geometry interface hierarchy of GeomLib. Rounded rectangles represent interfaces.

is against a point instead of a rectilinear curve, `leftRightTurn(Point2D,Point2D,Point2D)`, testing whether the three points form a left turn, a right turn, or are collinear, and `insideOutside(Circle2D,Point2D)`, testing whether the point is inside, on, or outside the circle.

For many interfaces of this subcomponent, various implementations are possible, according to which representation is chosen for the geometric objects. If we consider a point, for instance, we can represent its coordinates as integer numbers, exact rational numbers, or real number approximations; if we choose rational numbers, it may be convenient to adopt homogeneous coordinates rather than Cartesian ones; and other representations, differing not only for the type of arithmetic used, may be possible, as shown in Section 6.2.

We have grouped all the usual geometric tests in one interface in order to have their implementations localized, thus making their update easier in case a new representation of a basic geometric object should be added to the library. Ideally, geometric programs should never access directly the geometric information of the objects they manipulate, except for visualization purposes. All the geometric tests should be performed invoking the appropriate methods of the geometric tester interface. The important consequence of this approach is that the geometric program is completely independent from the representation chosen for the geometric objects and does not have to be changed if such representation changes. Provided that the results of the tests are correct for the chosen representation, the program will return correct results. This, of course, does not solve the crucial problem of how to guarantee the correctness of the geometric test results. We will discuss this fundamental issue in Section 4.4 and, with respect to our case study, in Section 6.2.

As a simple example of the above concepts, we present in Figs. 4 and 5 the GeomLib implementation of the Graham's scan algorithm for the two-dimensional convex hull problem. In particular, note how, in the auxiliary methods called form the main `grahamScan(Sequence)`, the geometric information of the points is never accessed directly. Objects of type `Point2D` are passed as parameters to methods of `GeomTester2D`. Each of these methods is implemented as a two-step procedure: determine what class implementing `Point2D` the parameters are instance of, and execute the code corresponding to that class. It is only in the implementation of the test methods that the geometric information of the parameters is accessed. The implementation of the Graham's scan algorithm, being written only in terms of interfaces, can safely ignore which class implementing `Point2D` is actually used. Note, also, that all possible inputs are considered, including degenerate ones, such as less than three points, collinear points, or points coincident with the anchor point.

4.3.2 The Advanced Geometric Subcomponent

In this subcomponent, some advanced geometric objects, such as *two-dimensional geometric graph* and *planar subdivision*, are defined. A two-dimensional geometric graph is a graph with the following associated geometric information: each vertex is mapped to a distinct two-dimensional point and each edge is mapped to a two-dimensional curve. Accordingly, the two-dimensional geometric graph interface inherits from both the graph interface and the two-dimensional geometric object interface. A planar subdivision is a two-dimensional geometric graph in which the underlying graph is an embedded planar graph, the two-dimensional curves mapped to the edges are pairwise non-intersecting, and the faces are mapped to two-dimensional (typically closed) curves. Accordingly, the planar subdivision inherits from both the embeddded planar graph interface and the two-dimensional geometric graph interface.

This subcomponent corresponds to the `jdsl.geom` package of JDSL. The present interfaces are the following:

```
// main method
public static Sequence grahamScan (Sequence points) {
  Point2D p1;
  Point2D p2;
  // copy into hull the sequence of input points
  copyInputPoints(points);
  switch (hull.size()) {
  case 0: case 1:
    return hull;
  case 2:
    p1 = (Point2D)hull.first().element();
    p2 = (Point2D)hull.last().element();
    if (geomTester.areEqual(p1,p2))
      hull.removeLast();
    return hull;
  default:  // at least 3 input points
    // compute the anchor point and remove it from hull together with all the points coincident with it
    anchorPointSearchAndRemove();
    switch (hull.size()) {
    case 0: case 1:
      hull.insertFirst(anchorPoint);
      return hull;
    default:  // at least 2 input points left besides the anchor point
      // sort the points in hull by polar angle w.r.t. the anchor point
      sortPoints();
      // remove the (possible) initial collinear points in hull except the farthest one from the anchor point
      removeInitialIntermediatePoints();
      if (hull.size() == 1)
        hull.insertFirst(anchorPoint);
      else {
        // insert the anchor point as first and last element in hull
        hull.insertFirst(anchorPoint);
        hull.insertLast(anchorPoint);
        // Graham's scan
        scan();
        // remove one of the two copies of the anchor point from hull
        hull.removeLast();
      }
      return hull;
    }
  }
}

// sort the points in hull by polar angle w.r.t. the anchor point
private static void sortPoints() {
  SortObject sorter = new ListMergeSort();
  ConvexHullComparator comp = new ConvexHullComparator(anchorPoint,geomTester);
  sorter.sort(hull,comp);
}

// private class variables
private static Sequence hull;
private static Point2D anchorPoint;
private static GeomSelector2D geomSelector = new GeomSelector2DImpl();
private static GeomTester2D geomTester = new GeomTester2DImpl();
```

Figure 4: The GeomLib implementation of the Graham's scan algorithm.

```
// compute the anchor point and remove it from hull together with all the coincident points
private static void anchorPointSearchAndRemove () {
  Enumeration pe = hull.positions();
  Position anchor = (Position)pe.nextElement();
  anchorPoint = (Point2D)anchor.element();
  // hull contains at least three elements
  while (pe.hasMoreElements()) {
    Position pos = (Position)pe.nextElement();
    Point2D p = (Point2D)pos.element();
    int aboveBelow = geomTester.aboveBelow(anchorPoint,p);
    int leftRight = geomTester.leftRight(anchorPoint,p);
    if (aboveBelow == GeomTester2D.BELOW ||
        aboveBelow == GeomTester2D.ON && leftRight == GeomTester2D.LEFT) {
      anchor = pos;
      anchorPoint = p;
    }
    else
      if (aboveBelow == GeomTester2D.ON && leftRight == GeomTester2D.ON)
        hull.remove(pos);
  }
  hull.remove(anchor);
}

// remove the (possible) initial collinear points in hull except the farthest one from the anchor point
private static void removeInitialIntermediatePoints() {
  boolean collinear = true;
  while (hull.size() > 1 && collinear) {
    Position pos1 = hull.first();
    Position pos2 = hull.after(pos1);
    Point2D p1 = (Point2D)pos1.element();
    Point2D p2 = (Point2D)pos2.element();
    if (geomTester.leftRightTurn(anchorPoint,p1,p2) == GeomTester2D.COLLINEAR)
      if (geomSelector.closer(anchorPoint,p1,p2) == p1)
        hull.remove(pos1);
      else
        hull.remove(pos2);
    else
      collinear = false;
  }
}

// Graham's scan
private static void scan() {
  Position first = hull.first();
  Position last = hull.last();
  Position prev = hull.first();
  Position curr = hull.after(prev);
  do {
    Position next = hull.after(curr);
    Point2D prevPoint = (Point2D)prev.element();
    Point2D currPoint = (Point2D)curr.element();
    Point2D nextPoint = (Point2D)next.element();
    if (geomTester.leftRightTurn(prevPoint,currPoint,nextPoint) == GeomTester2D.LEFT_TURN)
      prev = curr;
    else {
      hull.remove(curr);
      prev = hull.before(prev);
    }
    curr = hull.after(prev);
  }
  while (!hull.isLast(curr));
}
```

Figure 5: The GeomLib implementation of the Graham's scan algorithm (continued).

`GeomGraph2D` This interface describes a two-dimensional geometric graph and extends both `Graph` and `GeomObject2D`. Its main additional methods are `getPoint(Vertex)` and `getCurve(Edge)`, returning the geometric information associated with a vertex or an edge.

`PlanarSubdivision` This interface describes a planar subdivision and extends both `EmbeddedPlanarGraph` and `GeomGraph2D`. Its main additional method is `getCurve(Face)`, returning the geometric information associated with a face.

The planar subdivision is a good example of the advantages of the interface mechanism. Through interface multiple inheritance, the same implementation of the planar subdivision allows different views: simply as a container, ignoring the combinatorial, topological, and geometric information; as a combinatorial or topological graph, ignoring the associated geometric information; as a two-dimensional geometric graph, ignoring the topological information, or as a combination of topological graph and two-dimensional geometric graph, considering both the topological and geometric information.

4.4 The Arithmetic Component

In Section 4.3.1, we have seen how the encapsulation of the arithmetic properties within the basic geometric objects allows the implementation of a geometric algorithm to be independent from the arithmetic used. However, the problem of the correctness of the arithmetic computations has to be considered, as indicated, e.g., in [3, 8, 13, 17, 22, 23, 28, 32, 33, 34, 41, 58, 59, 67, 68]. The assumption of real number arithmetic has proved unrealistic, since digital computers do not exhibit such capability. On the other hand, exact rational arithmetic may excessively slow down computations. In light of these problems, the concepts of *degree* of a geometric algorithm [41] and of *arithmetic filter* [12] have been recently introduced. Informally, the degree of a geometric algorithm characterizes, up to a small additive constant, the arithmetic precision, i.e., the number of bits, required by the geometric algorithm. A more detailed discussion on the degree is given in Section 6.2. Since the degree of a geometric algorithm expresses worst-case computational requirements occurring in degenerate or near-degenerate instances, special attention must be devoted to the development of a methodology that reliably computes the sign of the algebraic expression corresponding to a geometric test, with the least expenditure of computational resources. This involves the use of arithmetic filters, possibly families of filters of progressively increasing power, that, depending upon the values of the test variables, carefully adjust the computational effort. We plan to implement arithmetic filters in the near future.

5 Design Evaluation

It is too early to judge our prototype against the requirements presented in Section 2, but we can evaluate the preliminary design:

Ease of use GEOMLIB interfaces are designed to be easy to use. We have tried to keep the number of methods for each interface low. In the choice of the method names we have preferred clarity to brevity.

Efficiency For each interface, different implementations are or will be provided. This will allow the users to choose the most appropriate one considering their efficiency constraints. For dynamic data structures, for instance, the user will be able to choose among different implementations presenting the usual trade-offs between query and update time.

A possible source of inefficiency arises from the choice of Java as the implementation language. Its cross-platform capability comes at the price of a reduced execution speed. However, in the near future the difference in execution speed between a Java program and a, say, C++ program should become insignificant thanks to the use of second-generation Java virtual machines, such as Sun's HotSpot, or the use of high performance compilers that produce optimized platform-specific native code, such as that developed by the alphaWorks team at IBM (see also Appendix B).

Reliability GeomLib will guarantee robust geometric computing, through the use of both exact rational arithmetic and floating point arithmetic with arithmetic filters. Also, particular attention will be placed, in the implementation of the algorithms, to handling all inputs, including the degenerate ones.

Openness In Section 4.3, we have described how the interface mechanism allows the users of GeomLib to choose the most appropriate representations of geometric objects for a given problem, without having to change the geometric program itself. We will show an example of use of this capability in Section 6.2. This approach seems to be more general of the one currently adopted in CGAL, where primitive geometric objects are parameterized, through the template mechanism, only w.r.t. the arithmetic type.

Extensibility The interface mechanism and the component-oriented design of GeomLib allow users to develop their own implementation of an existing interface and guarantee its interoperability with the rest of the library. At the same time, users will be able to include new interfaces and/or components.

Reusability We plan to maximize the internal reuse of code (two examples of internal reuse of code have been described in Sections 4.1.2 and 4.2).

Functionality Currently, there exists only a preliminary implementation of the JDSL/GeomLib design. In particular, it consists of the interfaces and classes necessary for the vertical case study described in Section 6. This should be contrasted with the much more advanced status of LEDA and CGAL. Our intention is to extend the implementation of GeomLib, possibly considering other case studies in different areas of computational geometry.

Modularity The architecture of GeomLib consists of a number of interrelated components. Combinatorial, topological, and geometric components describe properties of geometric objects at different levels of abstraction. The independence of the components (their *orthogonality*) is enforced by allowing them to interact only through a well-specified set of primitives.

Correctness Checking GeomLib does not currently contain correctness checkers, but will incorporate them in the near future in order to enhance reliability.

6 Point Location: A Vertical Case Study

In this section, we describe the design and implementation of a new algorithmic pattern, called the *binary space partition search*, that provides a unifying description of two well-known algorithms for the fundamental geometric problem of *planar point location*. Our goal is to provide a concrete example of how the innovative aspects of the GeomLib design allow conceptually simple implementations of geometric algorithms.

6.1 The Binary Space Partition Search Algorithmic Pattern

Abstraction has been, in the past years, a key technique for handling the increasing complexity of programs. The use of *algorithm abstraction* for geometric software has been recently proposed [63]. This term indicates the abstraction process aimed at the implementation of sophisticated algorithms as reusable software objects. It goes beyond both procedural abstraction (structured programming) and data abstraction (object-oriented programming), by viewing algorithms as objects that can be manipulated at the programming language level. It allows the programmer to construct new algorithms by specifying modifications and extensions of existing ones. It allows predefined algorithms to be combined in complex ways. These predefined algorithms of common use, called *algorithmic patterns*, are the main tools for algorithm abstraction. Examples of algorithmic patterns within the area of geometric computing include plane-sweep, lifting map, fractional cascading, and the binary space partition search that we are about to describe.

Planar point location is a fundamental search operation in computational geometry, and has been the target of substantial research. A survey on algorithms for planar point location is presented in [53]. We briefly recall the problem statement. Let S be a planar subdivision, with edges mapped to straight-line segments. S induces a partition of the Euclidean plane into a collection of *elementary* regions: open polygons (associated with the faces of S), open segments (associated with the edges of S), and points (associated with the vertices of S). Given a query point q, determine which region contains q. In the *repetitive mode* of operation, we can efficiently perform queries by preprocessing S into a suitable search structure. In the rest of the section, we denote by n the number of vertices of S.

We will first describe our design and implementation of a specific planar point location algorithm, the *chain method* [39, 16], and later show how the same design can be applied to another algorithm, the *trapezoid method* [52]. Both algorithms are very effective in practice, as reported in [14]. Finally, we will show how the binary space partition search can be used as an algorithmic component in a third algorithm, the *triangulation refinement method* [36]. These algorithms have in common a recursive decomposition of the search space of logarithmic depth.

6.1.1 The Chain Method

In this section, we will consider the chain method for planar point location by Lee and Preparata [39]. In particular, we will focus on its variant described in Chapter 11 of [15]. We recall that the search structure requires $O(n)$ space, can be constructed in $O(n \log n)$ time, and allows $O(\log^2 n)$ query time.

Planar subdivision S is assumed to be *monotone*, i.e., such that, for each face of S, its intersection with any horizontal line is connected. Note, however, that horizontal edges are allowed. Our design of the algorithm is based on the two following observations:

- The search region (initially the planar subdivision S) can be recursively decomposed by means of *separators*.
- Each separator can be viewed as a search region, possibly in a lower-dimensional space.

The basic operation in the search algorithm is the *discrimination* of the query point against the current separator, i.e., testing whether the query point is on one side or the other of the separator, or on the separator itself. According to the result of the discrimination, a new search region (a subregion of the current one) is searched. Eventually, the new search region will be an elementary region and the search will terminate. The discrimination of the query point against

a separator may require a single geometric test, or may require a secondary point location process in the separator and the discrimination against the result. In the chain method, the separators of the plane are monotone polygonal chains, called for brevity *chains*, and the elementary regions are faces. The separators of a chain are *horizontal intervals* and vertices, and the elementary regions are edges. Each horizontal interval corresponds to a *horizontal subchain*, that is a subchain whose edges are all horizontal. The separators of a horizontal subchain are vertices, and the elementary regions are edges and vertices.

The search algorithm can be conceptually modeled as an algorithmic pattern, called *binary space partition search*, characterized by the interaction of the following types of object:

Regions The objects in which the query point is searched.

Separators The objects against which the query point is discriminated. Note that separators are regions.

Discriminators The objects that perform the discrimination of the query point against a separator. Note that, in general, the query point may be discriminated against a separator either horizontally or vertically.

Search Statuses The objects that provide the next region to be searched, based on the result of the discrimination of the query point against the separator of the current region.

In the binary space partition search, the structural properties of the search space recursive decomposition are kept separate from the data structures used to implement it. The *separator tree*, *chain tree*, and *horizontal subchain tree* search structures are completely encapsulated in the search statuses.

The conceptual model of the algorithmic pattern is shown in Fig 6, using a formalism inspired by the Object Modeling Technique (OMT) [56]: rounded rectangles represent interfaces, rectangles represent classes, dashed lines connect classes to the interfaces they implement, and solid lines connect interfaces (classes) to the interfaces (classes) they extend. We have captured the main aspects of the algorithmic pattern in the five following Java interfaces. Note, again, that the design of JDSL does not directly depend on Java; its validity extends to C++ and other object-oriented languages.

`Region` This interface describes a region in which the location is performed (see Fig. 7). Its two method are `locate(Point2D)`, returning the face, edge, or vertex region containing the query point, and `internalRegion()`, returning the internal region of this.

`SeparableRegion` This interface extends the `Region` interface and describes a region that is divided by a separator (see Fig. 7). It defines the additional methods `separator()`, `discriminator()`, and `pointLocStatus()`, returning the separator, the discriminator, and the status associated with this.

`Separator` This interface describes the separator of a `SeparableRegion` (see Fig. 7). Its main methods are `horWhichSide(Point2D,GeomTester2D)` and `vertWhichSide(Point2D, GeomTester2D)`, returning the result of the horizontal and vertical discrimination of the query point against this.

`Discriminator` This interface describes the invoker of the appropriate (either horizontal or for vertical) discrimination through method `whichSide(Separator,Point2D)` (see Fig. 8).

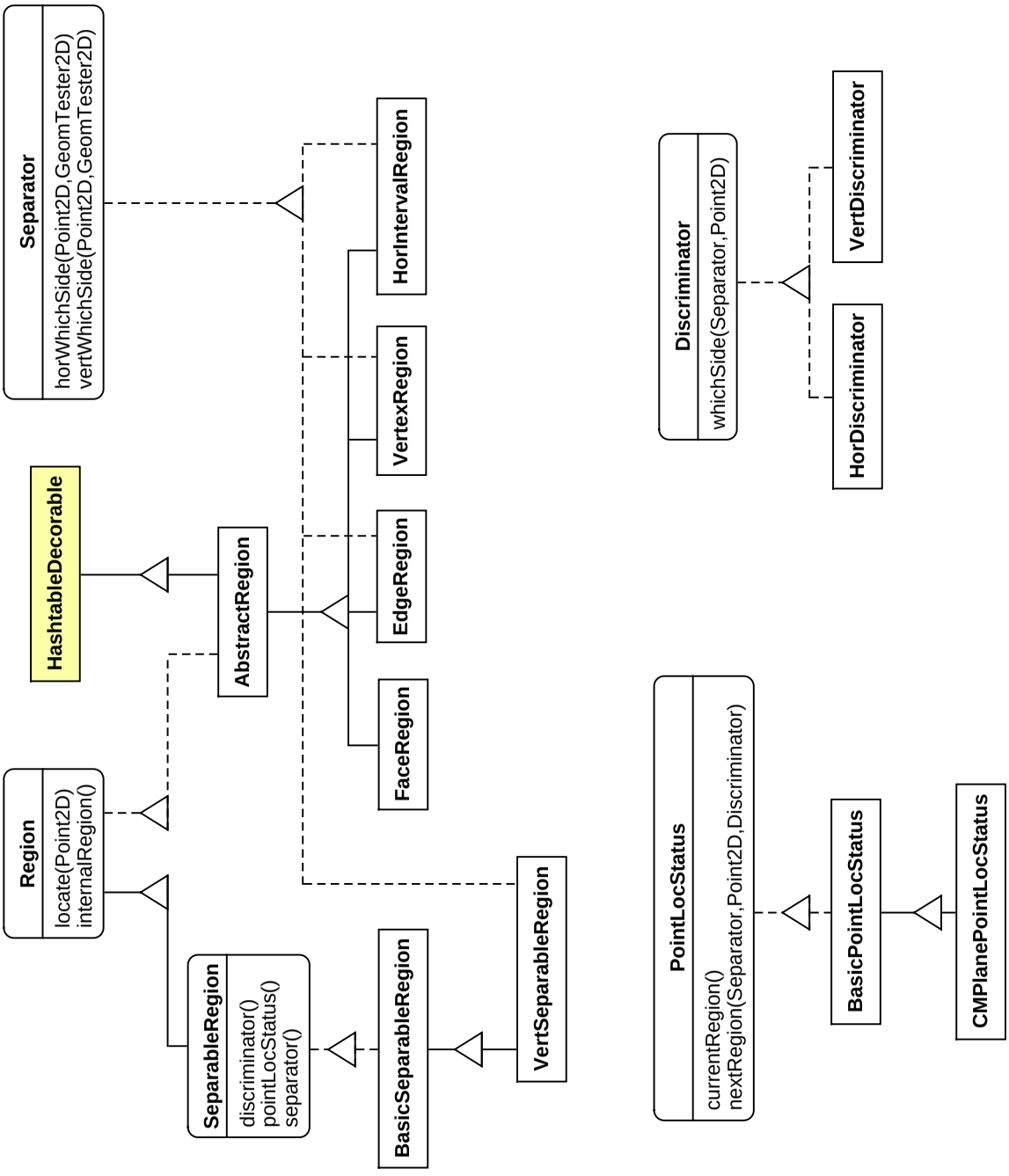

Figure 6: Interface and class hierarchies in the conceptual model for the binary space partition search algorithmic pattern. Rounded rectangles represent interfaces, rectangles represent classes, dashed lines connect classes to the interfaces they implement, and solid lines connect interfaces (classes) to the interfaces (classes) they extend.

```
/**
 * A region in which the location of a query point can be performed
 */
public interface Region {

  /**
   * @param q the query point
   * @return the region containing q
   */
  public Region locate (Point2D q);

  /**
   * @return the internal region associated with this
   */
  public Region internalRegion ();

}
```

```
/**
 * A region that is recursively divided in two by a separator
 */
public interface SeparableRegion extends Region {

  /**
   * @return the discriminator associated with this
   */
  public Discriminator discriminator ();

  /**
   * @return the point location status maintainer of this
   */
  public PointLocStatus pointLocStatus ();

  /**
   * @return the separator of this
   */
  public Separator separator ();

}
```

```
/**
 * The separator of a separable region
 */
public interface Separator {

  /**
   * @param q the query point
   * @param gt the geometric tester
   * @return whether q is to the GeomTester2D.LEFT of, GeomTester2D.ON, or
   * GeomTester2D.RIGHT of this
   */
  public int horWhichSide (Point2D q, GeomTester2D gt);

  /**
   * @param q the query point
   * @param gt the geometric tester
   * @return whether q is to the GeomTester2D.BELOW, GeomTester2D.ON, or
   * GeomTester2D.ABOVE this
   */
  public int vertWhichSide (Point2D q, GeomTester2D gt);

}
```

Figure 7: The interfaces `Region`, `SeparableRegion`, and `Separator`.

```
/**
 * A discriminator of the query point against a separator
 */
public interface Discriminator {

  /**
   * @param s a separator
   * @param q the query point
   * @return whether q is on the GeomTester2D.NEGATIVE side of s,
   * GeomTester2D.ON s, or on the GeomTester2D.POSITIVE side of s
   */
  public int whichSide (Separator s, Point2D q);

}
```

```
/**
 * The status of a point location search
 */
public interface PointLocStatus {

  /**
   * @return the current region
   */
  public Region currentRegion ();

  /**
   * @param s a separator
   * @param q the query point
   * @param d a discriminator
   * @return the next region according to the result of the discrimination
   * of q agains s
   */
  public Region nextRegion (Separator s, Point2D q, Discriminator d);

}
```

Figure 8: The interfaces `Discriminator` and `PointLocStatus`.

`PointLocStatus` This interface describes a search status, which keeps track of the status of the search in a separable region (see Fig. 8). Its two methods are `currentRegion()`, returning the region in which the location is taking place and `nextRegion(Separator,Point2D,Discriminator)`, returning the region in which to continue the search.

These interfaces are implemented by the following classes, one of which an abstract class. Abstract classes cannot be instantiated; their purpose is to factor out some common features of their subclasses in order to avoid code duplication.

`BasicSeparableRegion` implementing interface `SeparableRegion` (see Fig. 9). It is used to model a recursive decomposition of both the plane and the horizontal subchains. Note how an extensive use of recursion and polymorphism allows us to reduce the number of conditional control statements used in our code. The query method `locate()` can be expressed in a very compact way. The three instance variables, `separator_`, `discriminator_`, and `status_` store a reference to the separator, to the discriminator, and to the search status of the object, respectively.

`VertSeparableRegion` extending class `BasicSeparableRegion` and implementing interface `Separator` (see Fig. 10). It is used to model both a recursive decomposition of a chain and the separator of a `PlaneSeparableRegion`. Its only additional methods are those of interface `Separator`. In particular, note how method `horWhichSide(Point2D,GeomTester2D)` is implemented by first performing a location of the query point in this, and then invoking the method `horWhichSide(Point2D,GeomTester2D)` on the result.

`BasicPointLocStatus` implementing interface `PointLocStatus` (see Fig. 11). It is used to maintain the status in the search structure associated with a chain or a horizontal subchain. The method `nextRegion(Separator,Point2D,Discriminator)` is implemented by invoking

```
public class BasicSeparableRegion implements SeparableRegion {

  // public constructor

  public BasicSeparableRegion (Separator separator, Discriminator discriminator, PointLocStatus status) {
    separator_ = separator; discriminator_ = discriminator;
    status_ = status; label_ = "";
  }

  // public instance methods from SeparableRegion

  public Discriminator discriminator () {
    return discriminator_;
  }

  public PointLocStatus pointLocStatus () {
    return status_;
  }

  public Separator separator () {
    return separator_;
  }

  // public instance methods from Region

  public Region locate (Point2D q) {
    return status_.nextRegion(separator_q,discriminator_).locate(q);
  }

  public Region internalRegion () {
    return this;
  }

  // public instance methods from java.lang.Object

  public String toString () {
    return label_;
  }

  // protected instance variables

  protected Separator separator_;
  protected Discriminator discriminator_;
  protected PointLocStatus status_;
  protected String label_;

}
```

Figure 9: The class `BasicSeparableRegion`.

```
public class VertSeparableRegion extends BasicSeparableRegion implements Separator {

  // public constructor

  public VertSeparableRegion (Separator separator, Discriminator discriminator, PointLocStatus status) {
    super(separator,discriminator,status);
  }

  // public instance methods from Separator

  public int horWhichSide (Point2D q, GeomTester2D gt) {
    return ((Separator)locate(q)).horWhichSide(q,gt);
  }

  public int vertWhichSide (Point2D q, GeomTester2D gt) {
    throw new PointLocationException("Illegal test: vertWhichSide(Point2D,GeomTester2D)");
  }

}
```

Figure 10: The class `VertSeparableRegion`.

first the method `whichSide(Separator,Point2D)` of interface `Discriminator`, and then, based on the result, one of the three protected methods `negativeRegion(Separator)`, `onRegion(Separator)`, `positiveRegion(Separator)`. Note that the descent along the search tree is implemented in the first and third of these protected methods.

`CMPlanePointLocStatus` extending class `BasicPointLocStatus` (see Fig. 12). It is used to maintain the status in the search structure associated with the plane. It overrides the protected methods of `BasicPointLocStatus`, since the descent mechanism in the separator tree is different from that in the chain and horizontal subchain trees.

`AbstractRegion` extending class `HashtableDecorable` and implementing interface `Region` (see Fig. 13). `HashtableDecorable` is a class of the `jdsl.core` package of JDSL implementing the `Decorable` interface (see Section 4.1.1); thus, all the subclasses of `AbstractRegion` can be decorated.

`VertexRegion` extending class `AbstractRegion` and implementing interface `Separator` (see Fig. 13). It models a vertex of the planar subdivision. At the same time, it acts as a separator for a chain and for a horizontal subchain.

`EdgeRegion` extending class `AbstractRegion` and implementing interface `Separator` (see Fig. 14). It models an edge of the planar subdivision. At the same time, it acts as one of the constituents of a chain and of a horizontal subchain.

`HorIntervalRegion` extending class `AbstractRegion` and implementing interface `Separator` (see Fig. 14). It models a horizontal subchain of the planar subdivision. At the same time, it acts as a separator for a chain. When method `internalRegion()` is invoked on an instance of this class, a `BasicSeparableRegion` modeling a recursive decomposition of a horizontal subchain is returned.

`FaceRegion` extending class `AbstractRegion` (see Fig. 15). It models a face of the planar subdivision.

```
public class BasicPointLocStatus implements PointLocStatus {

  // public constructor

  public BasicPointLocStatus (BinaryTree tree) {
    tree_ = tree; reset_ = true;
  }

  // public instance methods

  public Region currentRegion () {
    if (tree_.isExternal(current_))
      return (Region)current_.element();
    else
      return ((Region)((SeparableRegion)current_.element()).separator());
  }

  public Region nextRegion (Separator separator, Point2D q, Discriminator discriminator) {
    Region result;
    if (reset_) {
      reset(); reset_ = false;
    }
    switch (discriminator.whichSide(separator,q)) {
    case GeomTester2D.NEGATIVE:
      result = negativeRegion(separator);
      reset_ = tree_.isExternal(current_);
      return result;
    case GeomTester2D.ON:
      result = onRegion(separator);
      reset_ = true;
      return result;
    default:   // case GeomTester2D.POSITIVE:
      result = positiveRegion(separator);
      reset_ = tree_.isExternal(current_);
      return result;
    }
  }

  // protected instance methods

  protected void reset () {
    current_ = tree_.root();
  }

  protected Region negativeRegion (Separator s) {
    current_ = tree_.leftChild(current_);
    return (Region)current_.element();
  }

  protected Region onRegion (Separator s) {
    return (Region)s;
  }

  protected Region positiveRegion (Separator s) {
    current_ = tree_.rightChild(current_);
    return (Region)current_.element();
  }

  // protected instance variables

  protected BinaryTree tree_;
  protected Position current_;
  protected boolean reset_;

}
```

Figure 11: The class `BasicPointLocStatus`.

```
public class CMPlanePointLocStatus extends BasicPointLocStatus {

  // public constructor

  public CMPlanePointLocStatus (BinaryTree tree, int minFaceNum, int maxFaceNum, Object CHAIN_NUMBER,
                                Object LEFT_FACE_TSN, Object RIGHT_FACE_TSN) {
    super(tree);
    initialMinFaceNum_ = minFaceNum; initialMaxFaceNum_ = maxFaceNum;
    CHAIN_NUMBER_ = CHAIN_NUMBER;
    LEFT_FACE_TSN_ = LEFT_FACE_TSN; RIGHT_FACE_TSN_ = RIGHT_FACE_TSN;
  }

  // protected instance methods

  protected void reset () {
    current_ = tree_.root();
    minFaceNum_ = initialMinFaceNum_;
    maxFaceNum_ = initialMaxFaceNum_;
  }

  protected Region negativeRegion (Separator separator) {
    Region r = ((SeparableRegion)separator).pointLocStatus().currentRegion();
    maxFaceNum_ = ((Integer)((AbstractRegion)r).get(LEFT_FACE_TSN_)).intValue();
    descend();
    return (Region)current_.element();
  }

  protected Region onRegion (Separator separator) {
    Region r = ((SeparableRegion)separator).pointLocStatus().currentRegion();
    return r.internalRegion();
  }

  protected Region positiveRegion (Separator separator) {
    Region r = ((SeparableRegion)separator).pointLocStatus().currentRegion();
    minFaceNum_ = ((Integer)((AbstractRegion)r).get(RIGHT_FACE_TSN_)).intValue();
    descend();
    return (Region)current_.element();
  }

  protected void descend () {
    boolean stop = false;
    while (!tree_.isExternal(current_) && !stop) {
      int cn = ((Integer)current_.get(CHAIN_NUMBER_)).intValue();
      if (cn > maxFaceNum_)
        current_ = tree_.leftChild(current_);
      else {
        if (cn <= minFaceNum_) {
          current_ = tree_.rightChild(current_);
          while (current_.element() == Container.NULL)
            current_ = tree_.leftChild(current_);
        }
        else
          stop = true;
      }
    }
  }

  // protected instance variables

  protected int initialMinFaceNum_, minFaceNum_;
  protected int initialMaxFaceNum_, maxFaceNum_;
  protected Object CHAIN_NUMBER_;
  protected Object LEFT_FACE_TSN_, RIGHT_FACE_TSN_;

}
```

Figure 12: The class `CMPlanePointLocStatus`.

```
public abstract class AbstractRegion extends HashtableDecorable implements Region {

  // protected constructor

  protected AbstractRegion (String label) {
    label_ = label;
  }

  // public instance methods

  public Region locate (Point2D q) {
    return this;
  }

  public Region internalRegion () {
    return this;
  }

  // public instance methods from java.lang.Object

  public String toString () {
    return label_;
  }

  // protected instance variable

  protected String label_;

}
```

```
public class VertexRegion extends AbstractRegion implements Separator {

  // public constructor

  public VertexRegion (Point2D point, String label) {
    super(label);
    point_ = point;
  }

  // public instance methods

  public int horWhichSide (Point2D q, GeomTester2D gt) {
    return gt.leftRight(point_,q);
  }

  public int vertWhichSide (Point2D q, GeomTester2D gt) {
    return gt.aboveBelow(point_,q);
  }

  // protected instance variable

  protected Point2D point_;

}
```

Figure 13: The abstract class `AbstractRegion` and the class `VertexRegion`.

```
public class EdgeRegion extends AbstractRegion implements Separator {

  // public constructor

  public EdgeRegion (LinearCurve2D linearCurve, String label) {
    super(label);
    linearCurve_ = linearCurve;
  }

  // public instance methods

  public int horWhichSide (Point2D q, GeomTester2D gt) {
    return gt.leftRight(linearCurve_,q);
  }

  public int vertWhichSide (Point2D q, GeomTester2D gt) {
    return gt.aboveBelow(linearCurve_,q);
  }

  // protected instance variable

  protected LinearCurve2D linearCurve_;

}
```

```
public class HorIntervalRegion extends AbstractRegion implements Separator {

  // public constructor

  public HorIntervalRegion (String label, Point2D leftPoint, Point2D rightPoint, Region internalRegion) {
    super(label);
    leftPoint_ = leftPoint; rightPoint_ = rightPoint; internalRegion_ = internalRegion;
  }

  // public instance methods

  public Region internalRegion () {
    return internalRegion_;
  }

  // public instance methods from Separator

  public int horWhichSide (Point2D q, GeomTester2D gt) {
    if (gt.leftRight(leftPoint_,q) == GeomTester2D.LEFT)
      return GeomTester2D.NEGATIVE;
    else
      if (gt.leftRight(rightPoint_,q) == GeomTester2D.RIGHT)
        return GeomTester2D.POSITIVE;
      else
        return GeomTester2D.ON;
  }

  public int vertWhichSide (Point2D q, GeomTester2D gt) {
    return gt.aboveBelow(leftPoint_,q);
  }

  // protected instance variable

  protected Point2D leftPoint_;
  protected Point2D rightPoint_;
  protected Region internalRegion_;

}
```

Figure 14: The classes `EdgeRegion` and `HorIntervalRegion`.

```
public class FaceRegion extends AbstractRegion {

  // public constructor

  public FaceRegion (String label) {
    super(label);
  }

}
```

```
public class HorDiscriminator implements Discriminator {

  // public constructor

  public HorDiscriminator (GeomTester2D gt) {
    gt_ = gt;
  }

  // public instance methods

  public int whichSide (Separator separator, Point2D q) {
    return separator.horWhichSide(q,gt_);
  }

  // protected instance variable

  protected GeomTester2D gt_;

}
```

```
public class VertDiscriminator implements Discriminator {

  // public constructor

  public VertDiscriminator (GeomTester2D gt) {
    gt_ = gt;
  }

  // public instance methods

  public int whichSide (Separator separator, Point2D q) {
    return separator.vertWhichSide(q,gt_);
  }

  // protected instance variable

  protected GeomTester2D gt_;

}
```

Figure 15: The classes `FaceRegion`, `HorDiscriminator`, and `VertDiscriminator`.

`HorDiscriminator`, `VertDiscriminator` implementing interface `Discriminator` (see Fig. 15). They implement method `whichSide(Separator,Point2D)` by invoking method `horWhichSide(Point2D,GeomTester2D)` and `vertWhichSide(Point2D,GeomTester2D)` of interface `Separator`, respectively.

The essence of the point location algorithm is described by the interplay between recursive method `locate(Point2D)` of class `BasicSeparableRegion` and method `nextRegion(Separator,`

Point2D,Discriminator) of class BasicPointLocStatus. A schematic representation of the possible objects returned by method nextRegion(Separator,Point2D,Discriminator) is shown in Figs. 16, 17, and 18. In particular, when invoked by a BasicSeparableRegion modeling a recursive decomposition of the plane, the object returned may be another BasicSeparableRegion modeling either a recursive decomposition of the plane or a recursive decomposition of a horizontal subchain, a FaceRegion, an EdgeRegion, or a VertexRegion (see Fig. 16). When invoked by a VertSeparableRegion modeling a recursive decomposition of a chain, the object returned may be another VertSeparableRegion modeling a recursive decomposition of a chain, a HorIntervalRegion, an EdgeRegion, or a VertexRegion (see Fig. 17). When invoked by a BasicSeparableRegion modeling a recursive decomposition of a horizontal subchain, the object returned may be another BasicSeparableRegion modeling a recursive decomposition of a horizontal subchain, an EdgeRegion, or a VertexRegion (see Fig. 18).

Note how the only classes that interact with the geometric component of JDSL/GeomLib are VertexRegion, EdgeRegion, and HorIntervalRegion. In particular, methods horWhichSide(Point2D,GeomTester2D) and vertWhichSide(Point2D,GeomTester2D) are the only ones that access the geometry of the planar subdivision. For an object of class VertexRegion or HorIntervalRegion, the associated basic geometric object against which the query point is discriminated is of type Point2D, and the horizontal or vertical discrimination is performed through methods leftRight(Point2D,Point2D) or aboveBelow(Point2D,Point2D) of interface GeomTester2D, respectively. For an object of class EdgeRegion, the associated basic geometric ob-

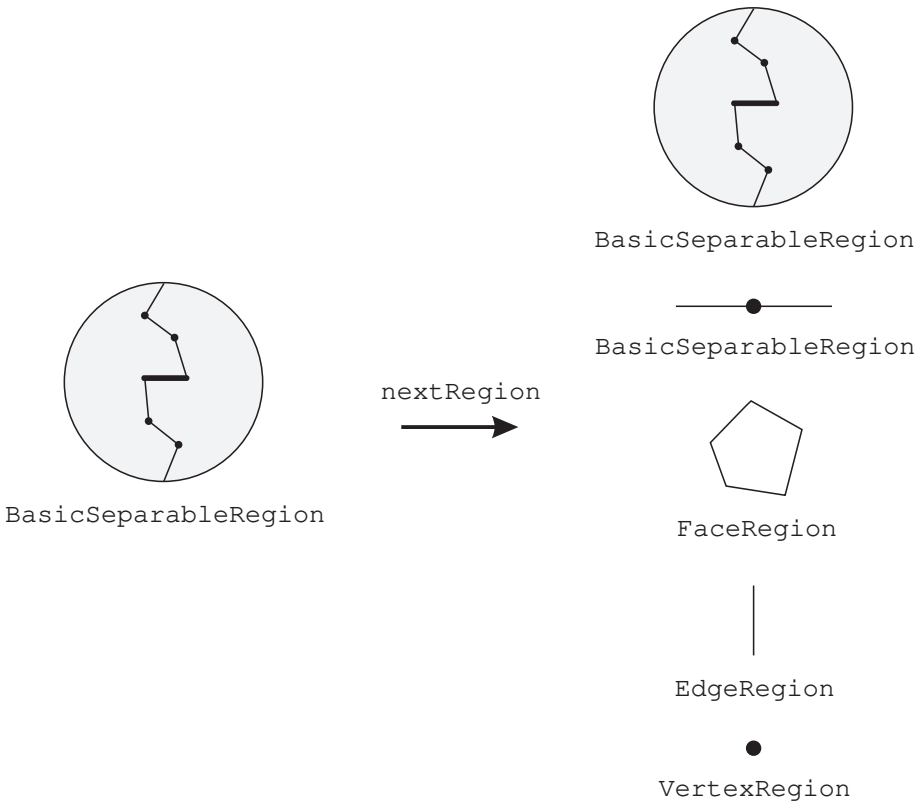

Figure 16: The possible objects returned by method nextRegion(Separator,Point2D,Discriminator) when invoked by a BasicSeparableRegion modeling a recursive decomposition of the plane.

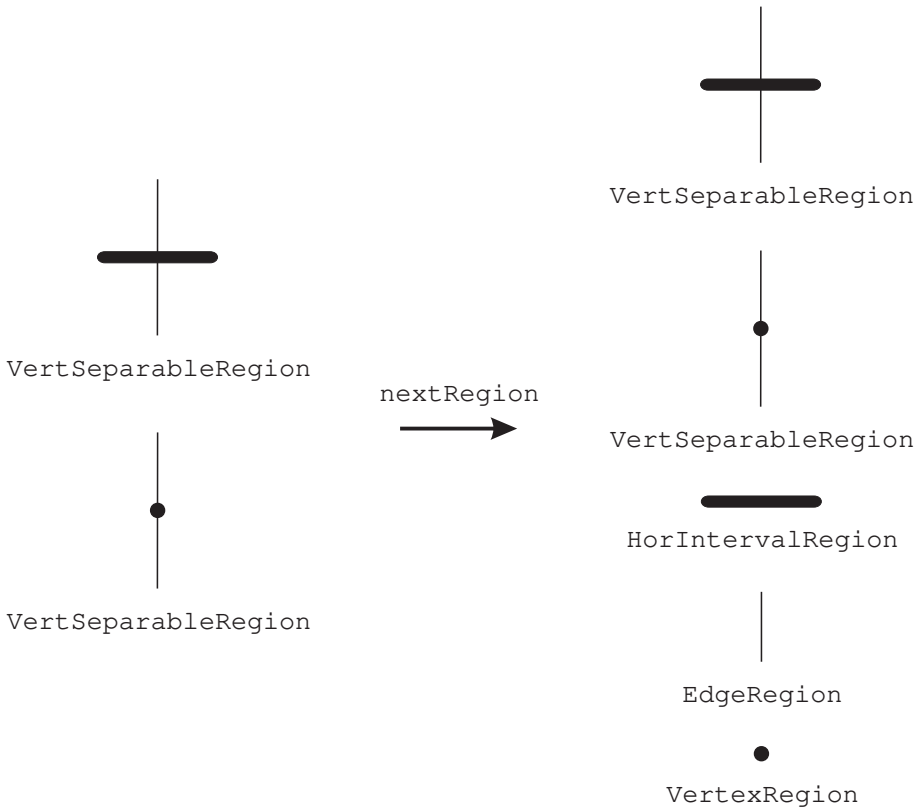

Figure 17: The possible objects returned by method `nextRegion(Separator,Point2D,Discriminator)` when invoked by a `VertSeparableRegion` modeling a recursive decomposition of a chain.

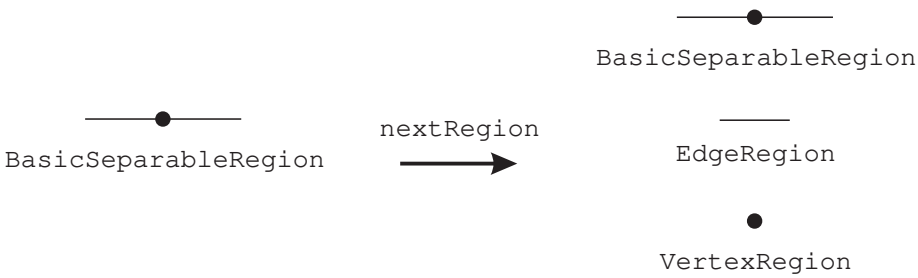

Figure 18: The possible objects returned by method `nextRegion(Separator,Point2D,Discriminator)` when invoked by a `BasicSeparableRegion` modeling a recursive decomposition of a horizontal subchain.

ject is of type `LinearCurve2D`, and the horizontal or vertical discrimination is performed through methods `leftRight(LinearCurve2D,Point2D)` or `aboveBelow(LinearCurve2D,Point2D)` of interface `GeomTester2D`, respectively.

In the implementation of the above classes, we make use of some basic data structures, such as sequences and binary trees. In particular, we use the two classes of the combinatorial component of JDSL, `NodeSequence` and `NodeBinaryTree`, which implement the interfaces `Sequence` and `BinaryTree`, respectively. For the plane, a `NodeBinaryTree` object is used to store the separator tree: each leaf of the separator tree contains a reference to a `FaceRegion` object, and each internal

node contains a reference to a `VertSeparableRegion` object. For each chain, a `NodeBinaryTree` object is used to store the corresponding chain tree: each leaf of the chain tree contains a reference to an `EdgeRegion` object, and each internal node contains a reference to either a `VertexRegion` or a `HorIntervalRegion` object. Finally, for each horizontal subchain, a `NodeBinaryTree` object is used to store the corresponding horizontal subchain tree: each leaf of the horizontal subchain tree contains a reference to an `EdgeRegion` object (except the first and last leaves), and each internal node contains a reference to a `VertexRegion` object.

6.1.2 The Trapezoid Method

In this section, we will briefly describe another planar point location algorithm, the trapezoid method of Preparata [52], through the same algorithmic pattern used for the chain method. We recall that, in the trapezoid method, the search structure requires $O(n \log n)$ space, can be constructed in $O(n \log n)$ time, and allows $O(\log n)$ query time.

The search space is recursively decomposed into regions, called *trapezoids*, by means of two types of separators: horizontal lines and edges of the planar subdivision. Again, the basic operation in the search algorithm is the discrimination of the query point against the current separator.

The binary space partition search is a correct abstraction for the trapezoid method. No new interface is required, and classes `BasicSeparableRegion`, `BasicPointLocStatus`, `VertexRegion`, `EdgeRegion`, `HorIntervalRegion`, `FaceRegion`, `HorDiscriminator`, and `VertDiscriminator` can be used without any modifications. In fact, when we create a `BasicSeparableRegion` object, we can specify an instance of a different class, w.r.t. those used in the chain method, as a `Separator` parameter, as long as the appropriate `Discriminator` parameter is passed. And thanks to the possibility, in Java, of using interfaces as object types, the implementation of the methods of `BasicSeparableRegion` need not be changed. In particular, in the chain method, the separator of a `BasicSeparableRegion` is either a `VertSeparableRegion` or a `VertexRegion`, while, in the trapezoid method, it is either a `VertexRegion`, or an `EdgeRegion`, or a `HorIntervalRegion`. Classes `VertSeparableRegion` and `CMPlanePointLocStatus` are not necessary in the trapezoid method.

Note that, even though the query algorithm for the chain and the trapezoid methods can be modeled through the binary space partition search algorithmic pattern, the preprocessing algorithms, are considerably different. In preprocessing the planar subdivision, one can even consider a mixed recursive decomposition of the search space: up to a certain level, the search space is decomposed, e.g., using chains, as in the chain method; after that, it is decomposed using, e.g., horizontal lines and edges, as in the trapezoid method.

6.1.3 The Triangulation Refinement Method

In this section, we will show how the binary space partition search can be used as an algorithmic component in another point location algorithm, the triangulation refinement method of Kirkpatrick [36]. We recall that, in this method, the search structure requires $O(n)$ space, can be constructed in $O(n \log n)$ time, and allows $O(\log n)$ query time.

The planar subdivision S is assumed to be a triangulation with exactly three external vertices. A sequence of triangulations with the same boundary is constructed, such that the first triangulation is the boundary of S, the $(k+1)$-th triangulation is a *refinement* of the k-th one, and the last triangulation is S itself. Each triangle in the k-th triangulation overlaps a set of adjacent triangles of the $(k+1)$-th triangulation. Thus, the search algorithm can be viewed as the repeated location of the query point in a sequence of sets of adjacent triangles. Since a set of adjacent triangles

is itself a planar subdivision, each location of the query point can be performed using the binary space partition algorithmic pattern.

However, the algorithm can also be easily implemented not using the binary space partition search algorithmic pattern as a component. All is needed is the definition of a new class `TriangleSetRegion` implementing interface `Region`. Method `locate` can then be implemented as a "brute force" inclusion test of the query point in all the triangles of the set.

In the triangulation refinement method, the sets of adjacent triangles are arranged in a directed acyclic graph search structure. In the combinatorial component of JDSL, we have defined the class `OnDemandGraph` implementing interface `Graph`. An `OnDemandGraph` object is used to represent the directed acyclic graph, where each vertex contains a reference to a `PlanarSubdivision` or to a `TriangleSetRegion` object.

6.2 Test Primitives in Voronoi Diagrams

In Section 6.1, we have shown that, for the binary space partition search algorithmic pattern, the only interaction between the topological component and the geometric component of JDSL is through methods `leftRight(Point2D,Point2D)`, `aboveBelow(Point2D,Point2D)`, `leftRight(LinearCurve2D,Point2D)`, and `aboveBelow(LinearCurve2D,Point2D)` of interface `GeomTester2D`. In this section, we will address a numerical precision issue in geometric computing, using the implementation details for these methods as an example. In particular, we will focus our attention on method `leftRight(LinearCurve2D,Point2D)`. In order to provide concreteness to our discussion, we will consider the case where the planar subdivision is a Voronoi diagram.

A lot of attention has been recently devoted, in the geometric computing community, to the problem of the robustness of geometric algorithms [3, 8, 13, 17, 22, 23, 28, 32, 33, 34, 41, 58, 59, 67, 68]. The real-RAM model, with its implicit infinite-precision requirement, has proved unrealistic and has to be replaced with a realistic finite-precision model where geometric computations can be carried out either exactly or with a guaranteed error bound. Also, the efficiency of the geometric algorithms must be evaluated in a finer framework than the conventional asymptotic analysis: in particular, constant factors dependent on the precision requirements of the numerical computations should be taken into account. Within the paradigm of exact computation (see, e.g., [3, 8, 67]), Liotta, Preparata, and Tamassia [41] have recently introduced the concept of *degree* of a geometric algorithm. The degree characterizes, up to a small additive constant, the arithmetic precision, i.e., the number of bits, required by a large class of geometric algorithms. Namely, if the coordinates of the input points of a d-degree algorithm within this class are b-bit integers, the algorithm may be required, on some instances, to perform computations with arithmetic precision $d(b + O(1))$. Since the arithmetic precision of a computation greatly affects the CPU time necessary to carry it out, the degree of a geometric algorithm should be considered as important as the asymptotic time complexity and should correspondingly play a major role in the design, or re-design, of a geometric algorithm. In [41], the problem area of geometric proximity is considered as a test case. In particular, the point location problem in a Voronoi diagram with applications to proximity queries is addressed, and a new efficient and low-degree technique for answering this type of query is presented.

We now briefly review the main ideas behind this new technique and show how the architecture of GeomLib allows us to implement them in an simple way. We assume that the coordinates of the input sites for the construction of the Voronoi diagram (also called *primitive* points) are b-bit integers. In a straightforward implementation of the point location algorithm, the vertices of the Voronoi diagram (also called *derived* points) are computed; we call the resulting diagram

an *explicit* Voronoi diagram. The derived points are rational numbers, and homogeneous coordinates (X, Y, W) can be used for their exact representation. However, coordinates X and Y are $3b$-bit integers, and coordinate W is a $2b$-bit integer. As shown in [41], this implies that, in an explicit Voronoi diagram, the discrimination of the query point against an edge (method `leftRight(LinearCurve2D,Point2D)` of interface `GeomTester2D` of GEOMLIB) has degree 6. The alternative approach proposed in [41] uses a different representation of the diagram, called *implicit* Voronoi diagram. It consists of a topological component and a geometric component. The topological component is the underlying embedded planar graph of the Voronoi diagram. The geometric component includes the following information: for each non-horizontal edge e of the diagram, the two primitive points $l(e)$ and $r(e)$ such that e is a portion of the perpendicular bisector of $l(e)$ and $r(e)$, and $l(e)$ is to the left of $r(e)$. With this representation, performing the discrimination of the query point against edge e only requires to compare the Euclidean distances of the query point from primitive points $l(e)$ and $r(e)$; this test has degree 2. As discussed in Section 4.3.1, the design of GEOMLIB allows geometric programs to be completely independent from the representation chosen for the geometric objects they handle. In this case, for instance, the explicit and implicit representations of a `LinearCurve2D` are interchangeable, without any need for modifications in the binary space partition search algorithmic pattern. The only requirement is that, in the implementation of interface `GeomTester2D`, each method taking a `LinearCurve2D` as a parameter, e.g., method `leftRight(LinearCurve2D,Point2D)`, have two different implementations, one for each representation.

7 Future Developments

We can divide the future developments into short-term and long-term ones. Short-term developments are aimed at completing the vertical case study. In particular, they include:

- The implementation of a regularizing algorithm for non-monotone planar subdivisions and the implementation of an algorithm for obtaining a monotone planar subdivisions from any (e.g., non-connected) planar subdivision.
- Experimental testing of planar point location methods has already been performed in the past [14]. We plan to use the prototype described in this paper to perform an experimental testing of point location queries in Voronoi diagrams, comparing the standard explicit representation of the diagrams with the implicit representation described in [41].
- The development of a collection of testers for the interfaces defined in JDSL/GeomLib. The purpose of these testers is to "certify" that a given class implements correctly an interface.
- The implementation of the checker for verifying the consistency of planar subdivisions described in [11].

Long-term developments are aimed at extending the current prototype of JDSL/GEOMLIB. In particular, they include:

- The design and implementation of other geometric algorithmic patterns, e.g., plane-sweep, space-sweep, and randomized incremental construction.
- The implementation of a system of arithmetic filters in the arithmetic component.

- The development of applets for the animation of geometric algorithms (see [30] for an introduction to this topic).
- To investigate the applicability of the binary space partition search algorithmic pattern to geographical information systems.

8 Acknowledgments

We would like to thank Franco Preparata and Mike Goodrich for their encouragement and support, Jim Baker for his collaboration in writing a preliminary version of this paper, Natasha Gelfand, Mark Handy, Benoit Hudson, David Jackson, and Masi Oka for many useful discussions about the design of JDSL/GeomLib and, together with Ryan Baker, Jitchaya Buranahirun, and Amit Sobti, for implementing and testing portions of the prototype.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 1996.

[2] J. Arvo, editor. *Graphics Gems II*. Academic Press, Boston, MA, 1991.

[3] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. Preparata, and M. Yvinec. Evaluating signs of determinants using single-precision arithmetic. *Algorithmica*, 17(2):111–132, 1997.

[4] M. Bertolotto, L. De Floriani, and P. Marzano. Unifying framework for the multilevel description of spatial data. In A. U. Frank and W. Kuhn, editors, *Spatial Information Theory (Proc. COSIT '95)*, volume 988 of *Lecture Notes Comput. Sci.*, pages 259–278. Springer-Verlag, 1995.

[5] M. Blum and S. Kannan. Designing programs that check their work. *J. ACM*, 42(1):269–291, 1995.

[6] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *J. Comput. System Sci.*, 47(3):549–595, 1993.

[7] C. Burnikel, J. Könnemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in LEDA. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C18–C19, 1995.

[8] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 16–23, 1994.

[9] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985.

[10] B. Chazelle et al. Application challenges to computational geometry: CG impact task force report. Technical Report TR-521-96, Princeton Univ., 1996.
`http://www.cs.princeton.edu/~chazelle/taskforce/CGreport.ps`.

[11] O. Devillers, G. Liotta, F. P. Preparata, and R. Tamassia. Checking the convexity of polytopes and the planarity of subdivisions. In F. Dehne, A. Rau-Chaplin, J.-R. Sack, and R. Tamassia, editors, *Algorithms and Data Structures (Proc. WADS '97)*, volume 1272 of *Lecture Notes Comput. Sci.*, pages 186–199. Springer-Verlag, 1997.

[12] O. Devillers and F. P. Preparata. A probabilistic analysis of the power of arithmetic filters. Technical Report CS-96-27, Center for Geometric Computing, Dept. Computer Science, Brown Univ., 1996.
`ftp://ftp.cs.brown.edu/pub/techreports/96/cs96-27.ps.Z`.

[13] D. P. Dobkin and D. Silver. Applied computational geometry: Towards robust solutions of basic problems. *J. Comput. Syst. Sci.*, 40(1):70–87, 1989.

[14] M. Edahiro, I. Kokubo, and T. Asano. A new point-location algorithm and its practical efficiency — Comparison with existing algorithms. *ACM Trans. Graph.*, 3(2):86–109, 1984.

[15] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1987.

[16] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15(2):317–340, 1986.

[17] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9(1):66–104, 1990.

[18] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic plane graph. *J. Algorithms*, 13(1):33–54, 1992.

[19] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung. Corrigendum (Maintenance of a minimum spanning forest in a dynamic plane graph). *J. Algorithms*, 15(1):173, 1993.

[20] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel: A basis for geometric computation. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry (Proc. WACG '96)*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 191–202. Springer-Verlag, 1996.

[21] U. Finkler and K. Mehlhorn. Checking priority queues. Manuscript, 1997.

[22] S. Fortune. Numerical stability of algorithms for 2D Delaunay triangulations. *Internat. J. Comput. Geom. Appl.*, 5(1&2):193–213, 1995.

[23] S. Fortune and C. J. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph.*, 15(3):223–248, 1996.

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

[25] N. Gelfand, M. T. Goodrich, and R. Tamassia. Teaching data structure design patterns. In *Proc. 29th ACM SIGCSE Tech. Sympos.*, 1998 (to appear).

[26] A. S. Glassner, editor. *Graphics Gems.* Academic Press, Boston, MA, 1990.

[27] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java.* John Wiley, New York, NY, 1998 (to appear).

[28] L. J. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 208–217, 1989.

[29] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4(2):74–123, 1985.

[30] A. Hausner and D. Dobkin. Making geometry visible: An introduction to the animation of geometric algorithms. In J.-R. Sack and J. Urrutia, editors, *Handbook on Computational Geometry.* North-Holland, Amsterdam, The Netherlands, 1998 (to appear).

[31] P. Heckbert, editor. *Graphics Gems IV.* Academic Press, Boston, MA, 1994.

[32] C. M. Hoffmann. The problems of accuracy and robustness in geometric computation. *IEEE Computer*, 22(3):31–41, 1989.

[33] C. M. Hoffmann, J. E. Hopcroft, and M. T. Karasick. Robust set operations on polyhedral solids. *IEEE Comput. Graph. Appl.*, 9(6):50–59, 1989.

[34] J. E. Hopcroft and P. J. Kahn. A paradigm for robust geometric algorithms. *Algorithmica*, 7(4):339–380, 1992.

[35] D. Kirk, editor. *Graphics Gems III.* Academic Press, Boston, MA, 1992.

[36] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[37] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing.* Addison Wesley, Reading, MA, 1993.

[38] M. Laszlo. *Computational Geometry and Computer Graphics in C++.* Prentice-Hall, Englewood Cliffs, NJ, 1996.

[39] D. T. Lee and F. P. Preparata. Location of a point in a planar subdivision and its applications. *SIAM J. Comput.*, 6(3):594–606, 1977.

[40] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, MA, 1997.

[41] G. Liotta, F. P. Preparata, and R. Tamassia. Robust proximity queries: An illustration of degree-driven algorithm design. *SIAM J. Computing*, to appear.
`http://www.cs.brown.edu/cgc/papers/lpt-rpqid-.ps.gz`.

[42] M. C. Loui et al. Strategic directions in research in theory of computing. *ACM Comput. Surv.*, 28(4):575–590, 1996.

[43] K. Mehlhorn, M. Müller, S. Näher, S. Schirra, M. Seel, C. Uhrig, and J. Ziegler. A computational basis for higher-dimensional computational geometry and applications. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 254–263, 1997.

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

[45] K. Mehlhorn, S. Näher, T. Schilz, S. Schirra, M. Seel, R. Seidel, and C. Uhrig. Checking geometric programs or verification of geometric structures. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 159–165, 1996.

[46] K. Mehlhorn, S. Näher, and C. Uhrig. The LEDA platform for combinatorial and geometric computing. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Automata, Languages and Programming (Proc. ICALP '97)*, volume 1256 of *Lecture Notes Comput. Sci.*, pages 7–16. Springer-Verlag, 1997.

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

[48] ObjectSpace. JGL, the generic collection library for Java. `http://www.objectspace.com/JGL`.

[49] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, Cambridge, England, 1994.

[50] M. H. Overmars. Designing the Computational Geometry Algorithms Library CGAL. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry (Proc. WACG '96)*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 53–58. Springer-Verlag, 1996.

[51] A. W. Paeth, editor. *Graphics Gems V.* Academic Press, Boston, MA, 1995.

[52] F. P. Preparata. A new approach to planar point location. *SIAM J. Comput.*, 10(3):473–482, 1981.

[53] F. P. Preparata. Planar point location revisited. *Internat. J. Found. Comput. Sci.*, 1(1):71–86, 1990.

[54] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.

[55] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, Cambridge, England, 2nd edition, 1993.

[56] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and J. Rumbaugh. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[57] S. Schirra. Designing a computational geometry algorithms library. Lecture Notes for Advanced School on Algorithmic Foundations of Geographic Information Systems, CISM, Udine, Italy, September 1996. `http://www.cs.ruu.nl/CGAL/Information/Papers/udine96-ss.ps.gz`.

[58] J. R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, 1996.

[59] K. Sugihara and M. Iri. A robust topology-oriented incremental algorithm for Voronoi diagrams. *Internat. J. Comput. Geom. Appl.*, 4(2):179–228, 1994.

[60] A. Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479, 1996.

[61] R. Tamassia. On-line planar graph embedding. *J. Algorithms*, 21(2):201–239, 1996.

[62] R. Tamassia et al. Strategic directions in computational geometry. *ACM Comput. Surv.*, 28(4):591–606, 1996.

[63] R. Tamassia, M. T. Goodrich, S. P. Reiss, and Y. Amir. A software platform and network environment for geometric computing. Manuscript, Center for Geometric Computing, 1996.

[64] P. van Oosterom. *Reactive Data Structures for Geographic Information Systems.* Oxford University Press, Oxford, England, 1993.

[65] R. C. Veltkamp. Generic programming in CGAL, the Computational Geometry Algorithms Library. In F. Arbabm and P. Slusallek, editors, *Proc. 6th Eurographics Workshop on Programming Paradigms in Graphics*, 1997.

[66] P. Wegner. Concepts and paradigms of object-oriented programming. *ACM OOPS Messenger*, 1(1):7–87, 1990.

[67] C. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7(1):3–23, 1997.

[68] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 1 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific Press, Singapore, 2nd edition, 1995.

A Object-Oriented Concepts

The design of GeomLib relies on various object-oriented design concepts (see, e.g., [9, 24, 60, 66]). In this section, we will review the most relevant ones.

Object At the conceptual level, an object is an abstraction describing all relevant aspects of a certain entity of the reality to be modeled (a particular application domain or the solution space of a particular problem).

At the implementation level, an object consists of an internal state and a collection of operations. The operations, usually called *methods*, represent the only way to interact with the object, i.e., accessing and possibly modifying its internal state. In other words, the internal state is hidden to other objects, it cannot be accessed and modified directly; this property is called *encapsulation*. Variables used for storing the internal state are called *instance variables*.

Objects have an associated *identity*, which makes it possible to distinguish one object from all the others and allows all references to the same object to be recognized as equivalent. An object's identity is logically distinct from its state or behavior, since the latter are not unique.

Class At a conceptual level, a class is an abstraction used to describe the common aspects of a set of objects.

At the implementation level, a class serves as template from which objects can be created. It contains a definition of the instance variables and of the methods for its objects. The act of creating an object of a certain class is called *instantiation*, and the object is called an *instance* of the class. Two instances of a class have private copies of the instance variables defined in the class while they share the methods.

One may think of a class as specifying a behavior common to all its instances: the methods determine the behavior, while the instance variables specify a structure for realizing it.

A class acts as a type for its instances, allowing object type checking.

Interface An interface is a description of the behavior of a class of objects, regardless of the representation chosen for the class. This description usually consists in the declaration of all the methods of the class. Interfaces can be seen as a contract between classes of objects and their users. However, since the state of the object is not modeled in the interface, it is not possible to use an interface to define a protocol of interaction between an object and its users. This task is accomplished, as seen above, by classes.

A class is said to *implement* an interface if it conforms to the behavior described in the interface, i.e., if it provides the semantics for it. This is accomplished by providing an actual implementation of all the methods declared in the interface. (Note the different, yet related, meanings of the verb "to implement".) Different classes may implement the same interface (in different ways). Also, a class may implement different interfaces at the same time, allowing an object to be of multiple, orthogonal types.

Like classes, interfaces can be used as a type for objects. This approach presents the following advantage: an object whose type is interface I can refer to an instance of any class implementing I.

Inheritance Inheritance is a mechanism that allows the definition of a new class or interface to be based upon that of some existing class or interface, respectively. Accordingly, we will distinguish between *class inheritance* and *interface inheritance*.

When a new class (interface) is to be defined, only the properties that differ from those of the specified existing classes (interfaces) need to be defined or redefined explicitly; the other properties are automatically extracted from the existing classes (interfaces) and included in the new class (interface). The new class (interface) is called a *subclass* (*subinterface*) of the specified existing classes (interfaces); each specified existing class (interface) is called a *superclass* (*superinterface*) of the new one. Each instance of a subclass is also an instance of its superclass(es). Inheritance relationships are transitive, and the term *class (interface) hierarchy* is generally used.

If a class (interface) can have only one superclass (superinterface), the term *single inheritance* is used; if, on the contrary, a class (interface) can have more than one superclass (superinterface) the term *multiple inheritance* is used.

Although similar, class inheritance and interface inheritance have different purposes. Class inheritance is an implementation mechanism for sharing behavior and data. Interface inheritance, on the other hand, is a conceptual mechanism for expressing generalization/specialization abstractions, i.e., for allowing new concepts to be derived from less specific ones. Thus, interface inheritance complements classification/instantiation abstraction provided by classes: classification is concerned with encapsulating the implementation details, generalization is concerned with composition and incremental modifications of already abstract behaviors.

Polymorphism Polymorphism is the ability for a programming language construct to have different types or to manipulate objects of different types. Various kinds of polymorphism have been defined; we briefly review the taxonomy described in [9].

Coercion This term indicates a mapping between different types (e.g., between type integer and type real) in a programming language.

Overloading This term indicates the possibility of defining multiple methods with the same name, either in the same class (interface) or in different classes (interfaces).

If methods with the same name are defined in the same class (interface), they must differ in the type of parameters and/or in the number of parameters.

If methods with the same name, type of parameters and number of parameters are defined in two classes (interfaces), one inheriting from the other, the term *overriding* is also used.

Inclusion polymorphism It allows a method to operate with parameters from a range of types. The range of types is determined by an inheritance relationship. If a certain class (interface) T is used as the type for a formal parameter, then any object whose type is a subclass (subinterface) of T can be passed to the method as the actual parameter. Note the difference between overloading and inclusion polymorphism: in the former case, the different definitions of the method correspond to different implementations; in the latter case, it is the same implementation of the method that can be used for different types of parameters.

Parametric polymorphism It allows a method to be defined with parameters of *generic* type. For each application of the method, the generic types for the parameters are replaced

with actual types (see footnote on page 5).

In order to fully take advantage of polymorphism, the binding of a method name to the code implementing it should be done at run time rather than at compile/link time; the term *dynamic binding*, as opposed to *static binding*, is used.

Also, type compatibility in assignment operations and parameter passing may be checked at compile/link time or at run time; the terms *static type checking* and *dynamic type checking* are used, respectively.

Design pattern A design pattern abstracts, identifies, and names the key aspects of a solution to a common object-oriented design problem. It identifies the classes and instances participating in the solution, their roles and the way they interact with each other. Collections of systematically described design patterns (see e.g., [24]) are a valuable tool for software designers. Their goal is to encourage the reuse of existing design solutions for similar problems rather than designing a new application from scratch. They are usually classified into three categories, according to their purpose:

Creational patterns concern the process of object creation.

Structural patterns deal with the composition of classes and objects.

Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

B Implementation Language

We have chosen Java as the implementation language for JDSL/GeomLib since it directly provides a number of concepts and language constructs used in object-oriented modeling and design, such as packages, interfaces, classes, objects, and polymorphism [1]. Note, however, that the JDSL/GeomLib design is not directly dependent on Java. We now briefly review some of those concepts and constructs:

Package It provides modularity by defining the name spaces for interfaces, classes, and objects; this modularity is also used to encapsulate entities within a package through name access controls. (It would be a security failure to allow untrusted packages to access trusted packages directly.) The naming of packages should conform to the Internet domain naming scheme to guarantee uniqueness of packages.

Visibility level Interfaces, classes, instance variables, class variables, and consequently objects, are protected from inappropriate access by different visibility levels (private, protected, package, and public). Although geometric libraries do not have the stringent security requirements of Internet browsers or digital payment systems, they can take advantage of encapsulation to help prevent incorrect usage and avoid unnecessary dependencies between objects.

Interface It contains a set of methods (at public or package visibility level) declared with their parameter types, return type, and exceptions thrown. An interface can also contain a set of constants. Interfaces can participate in multiple inheritance. Since they are essentially named sets, they can be trivial (empty) or the union of other interfaces. These capabilities amplify their use for typing purposes.

Class, *Object* See Appendix A. Classes can participate only in single inheritance.

Abstract class A partially implemented class which is supposed to be extended by other classes; it cannot be instantiated. Its main purpose is to factor out some common features of their subclasses in order to avoid code duplication.

Java compilers produce an intermediate byte code that is comparable to the intermediate code produced by most compilers, and is language independent. (Indeed a number of compilers have been introduced for other languages to produce Java byte code.) A number of aspects distinguishes this byte code from similar formats: it is portable, to the extent it is designed to load over a network, and it is secure. In particular, memory addresses and integers are distinct in Java byte code to prevent any type of pointer arithmetic. The byte code is loaded and executed by a Java virtual machine [1, 40].

Running a Java program with high performance usually involves converting the portable and secure intermediate Java byte code to platform-specific native code. The conversion may be executed, while the program is loading or running, by a just-in-time compiler, or, separately, by a high performance compiler, like the one developed by the alphaWorks team at IBM. Recently, however, Sun has announced a second-generation Java Virtual Machine, named HotSpot, which should allow Java byte code to be executed at a speed comparable to that of platform-specific native code.