A Case Study in Algorithm Engineering for Geometric Computing[*][†]

Roberto Tamassia
Center for Geometric Computing
Department of Computer Science
Brown University
Providence, Rhode Island 02912–1910, USA
rt@cs.brown.edu

Luca Vismara
Center for Geometric Computing
Department of Computer Science
Brown University
Providence, Rhode Island 02912–1910, USA
lv@cs.brown.edu

Abstract

The goal of this paper is to prove the applicability of algorithm engineering and software design concepts to geometric computing through a vertical case study on the implementation of planar point location algorithms. The work is presented within the framework of the GeomLib project, aimed at developing an easy to use, reliable, and flexible 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.

Keywords: Algorithm engineering, geometric computing, software libraries, point location.

[*]Research supported in part by the U.S. Army Research Office under grant DAAH04–96–1–0013 and by the National Science Foundation under grants CCR–9423847, CCR–9732327, and CDA-9703080. Portions of the results described in this paper were presented at the *Workshop on Geometric Computing*, Sophia Antipolis, France, 1997, and at the *1st Workshop on Algorithm Engineering (WAE '97)*, Venezia, Italy, 1997.

[†]This is a preprint of an article accepted for publication in the *International Journal of Computational Geometry & Applications* ©World Scientific Publishing Company.

1 Introduction

This paper overviews the preliminary design of the GeomLib library of geometric data structures and algorithms and presents a case study on the implementation of planar point location algorithms, which illustrates several key aspects of GeomLib and its underlying algorithm engineering principles.

1.1 Goals of GeomLib

The GeomLib project addresses the important objective of developing an easy-to-use, reliable, and flexible library of robust and efficient geometric algorithms. GeomLib is part of a larger project, named JDSL [36, 39, 40, 41], aimed at constructing a library of data structures and algorithms using the Java programming language.[1] In the design of GeomLib, we have taken into account the experience of related efforts, such as the Library of Efficient Data structures and Algorithms (LEDA) [13, 59, 60, 61] and the more recent Computational Geometry Algorithms Library (CGAL) [31, 68].

The main goals of the GeomLib project are:

- 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 algorithms available to users in major application areas, such as robotics, geographic information systems, mechanical engineering, and computer graphics.

Addressing the geometric computing requirements of application areas is a strategic direction for computational geometry [17, 58, 82]. In particular, the development of libraries of geometric data structures and algorithms is motivated by 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.

1.2 Key Aspects of GeomLib

Object-oriented software design concepts, such as design patterns [35] are extensively used throughout GeomLib. In particular, GeomLib defines a collection of interfaces[2] that describe some fundamental data structures used in geometric computing, suitably arranged in hierarchies. For each interface, different implementations are provided. This enables users to choose the most appropriate implementation considering efficiency constraints. Users of GeomLib are encouraged to develop algorithms using interfaces rather than specific implementations, thus creating more general code.

[1] http://www.cs.brown.edu/cgc/jdsl/.

[2] The term *interface* is used here to denote the declaration of the methods of a class of objects, and is similar to the concept of abstract data type.

In the design of JDSL/GeomLib, the combinatorial, topological, and geometric properties of geometric objects are accessed through different, increasingly specialized interfaces. For example, the interface of a planar subdivision specializes that of an embedded planar graph by adding methods that access geometric information (e.g., the point associated with a vertex and the segment associated with an edge), and in turn the interface of an embedded planar graph extends the interface of a graph with methods that access topological information (e.g., the cycle associated with a face and the clockwise ordering of the edges incident with a vertex). An advantage of this hierarchical design is that any combinatorial algorithm for graphs and any topological algorithm for embeddings can be executed directly (without adaptation) on a planar subdivision.

Abstraction is a key technique for handling the increasing complexity of programs. In the design of GeomLib, *algorithm abstraction* plays a major role. We call algorithm abstraction the process of implementing 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.

Using algorithm abstraction, one can construct reusable software components that embody algorithmic techniques of common use, such as plane-sweep, lifting map, fractional cascading, and binary space partition search (discussed in detail in this paper). We call *algorithmic pattern* the implementation of a specific algorithmic technique using algorithm abstraction. An algorithmic pattern provides the implementation of an algorithm template that can be specialized and combined to generate more complex computations. An algorithmic pattern is a refinement of the design pattern known as *template method* pattern [35]. Chapter 15 of Ref. [26] overviews several algorithmic techniques used in computational geometry that could be usefully implemented as algorithmic patterns.

A variety of techniques have been designed to make geometric algorithms robust in the presence of high-precision numerical computations (e.g., involving square roots) and degenerate geometric configurations (e.g., more than two collinear points or more than three cocircular points) [3, 14, 24, 28, 33, 34, 43, 47, 48, 49, 57, 76, 78, 85, 86, 87]. GeomLib adopts the paradigm of exact computation (see, e.g., Refs. [3, 14, 86]) and uses the concept of *degree* [57] to characterize the arithmetic precision requirement of a geometric algorithm. Namely, a geometric algorithm of degree d requires in its computations a precision that is, in the worst case, about d times that of the input data. 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 (see, e.g., Refs. [6, 7, 20]).

As any library, GeomLib 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, the gap in execution speed between a Java program and, say, a C++ program is expected to narrow thanks to the use of second-generation Java virtual machines or high-performance compilers that produce optimized platform-specific native code.

1.3 A Vertical Case Study on Planar Point Location

We exemplify the main aspects of the design of GeomLib through the detailed discussion of a vertical case study on the design and implementation of an algorithmic pattern that unifies two well-known techniques for planar point location: the chain method (also known as separator method) and the trapezoid method. Exploiting the commonalities between these two methods allows us to reduce the amount of code necessary to implement them, providing a concrete example of fast prototyping of two complex geometric algorithms.

Planar point location is a fundamental search primitive in computational geometry. The problem is to preprocess a planar subdivision S with n vertices in order to efficiently support queries that find the region of S containing a query point. Several efficient point location data structures have been devised (see, e.g., the surveys [72, 77]). Most of these data structures use a common scheme, which we call *binary space partition search*, consisting of: *(i)* a recursive decomposition of the planar subdivision by means of separators, which can be atomic geometric objects, such as segments or horizontal lines, or more complex structures, such as monotone polygonal chains; *(ii)* the representation of the decomposition structure by means of a binary tree T. Each internal node of T is associated with a portion of S and with a separator. A point location search then proceeds by traversing a path from the root of T (associated with the entire subdivision S) down to a leaf (associated with a region of S) determining, at each visited node μ, on which side of the separator associated with μ lies the query point. By balancing the decomposition tree T, efficient query time can be obtained.

The popular *chain method* [27, 55] and *trapezoid method* [71] follow the binary space partition search scheme. Both techniques are very efficient in practice, as reported in Ref. [25]. Also, by extending the chain method with fractional cascading, one can obtain a theoretically optimal point location data structure with $O(n)$ space requirement and $O(\log n)$ query time [27].

In this paper, we show how the binary space partition search scheme can be developed into an algorithmic pattern by implementing within GeomLib a reusable software component for planar point location, and demonstrate how the chain method and trapezoid method can be obtained by a simple specialization of this component.

1.4 Organization of the Paper

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 the Standard Template Library, 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. Concluding remarks and 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, in Appendix B we motivate the choice of Java as the implementation language for the current JDSL/GeomLib prototype, and in Appendix C we give the code fragments for some of the classes defined in Section 6.

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 others are particularly important for geometric computing.

Ease of Use The library should be easy to use. This requirement depends on various factors that, in general, differ for different categories of users. We recall some of the most relevant. The number of concepts used in the library should be minimal and guarantee an adequate functionality. An appropriate level of abstraction for these concepts is critical in this regard. The naming scheme adopted for components and operations should be intuitive and consistent throughout the library. The documentation should be integrated into the library. Knuth pioneered integration between code and documentation with the concept of "literate programming" (see, e.g., Ref. [53]). Note that the Java programming language provides this capability as part of its specification [1]. All this should contribute to a smooth learning curve for the users of the library.

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 *precision* [10], the *degree* [57], or the *depth of derivation* [86, 87].

Flexibility 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., Ref. [83]).

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 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 [31, 75, 82]. Note, however, that a trade-off exists between reliability and other criteria, in particular efficiency.

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. 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. Modular architectures should be contrasted with *monolithic* ones.

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., Refs. [4, 5, 22, 32, 62]. 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 Ref. [62], 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, STL, Java Collections and JGL, and the European initiatives LEDA and CGAL. We evaluate previous work according to the requirements presented in Section 2. For the gems/recipes approach some of the requirements are not particularly meaningful and we omit them.

3.1 Gems and recipes

"Graphics gems" [2, 38, 46, 52, 69] and "numerical recipes" [73] have proved very useful in computer graphics and scientific computing, respectively. Through books and available source code, the gems/recipes approach has disseminated easy to use, 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,[3] created by Nina Amenta, 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.[4] Almost all large libraries provide some degree of correctness checking, through a validation suite, and of flexibility.

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 lack modularity, as typical for libraries written in Fortran.

Some small specialized libraries have been written for computational geometry, usually accompanying introductory textbooks (see e.g., Refs. [54, 67]). They are typically easy to use, but lack functionality.

3.3 STL, Java Collections, and JGL

The Standard Template Library [65, 70], or STL, is a C++ library of containers, iterators, algorithms, and function objects. It is part of the C++ ISO standard, and various implementations are available. The STL is a *generic* library, meaning that its components are heavily parameterized: almost every component in the STL is a template.[5] STL provides many basic data structures, such as vectors, lists, sets, maps, hash tables, and priority queues, collectively referred to as *containers*. It also includes a large collection of algorithms that manipulate the data stored in containers. According to the principles of generic programming, algorithms are decoupled from the container classes they operate on; they are not methods of container classes, but rather global methods. The decoupling of algorithms from containers is made possible by *iterators*. Iterators are a generalization of pointers, and, as the name suggests, are often used to iterate over a range of objects: if an iterator points to one element in a range, then it is possible to increment (decrement) it so that it points to the next (previous) element. Iterators are central to generic programming because they are an interface between containers and algorithms: algorithms typically take iterators as template arguments, so a container need only provide a way to access its elements using iterators. This makes it possible to write generic algorithms that operate on many different kinds of containers.

[3] http://www.geom.umn.edu/software/cglist/.

[4] http://www.netlib.org/.

[5] *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 template class can be defined with the notation `template <class T> class Set`, and later specialized to a set of integers class with the notation `Set<int>`.

Ease of Use Users of STL may need some time to fully take advantage of the generic programming principles on which the library is based. The inclusion of the library in the C++ standard, however, will result in a wider dissemination of these principles.

Efficiency In the design of STL, particular care has been taken as to the efficiency of the various operations and algorithms.

Flexibility The template mechanism on which STL is based provides a high degree of flexibility, especially when combined with the use of function objects.

Modularity STL has a flat design, not organized in architectural modules. The distinction of the various components of the library in containers, iterators, algorithms, and function objects is more conceptual than architectural.

Functionality STL provides many basic data structures and algorithms. It does not contain, however, more complex data structures such as trees, graphs, and planar subdivisions.

Two libraries similar to STL have been designed for Java, namely the `java.util` package of the Java Development Kit (JDK) by Sun [79], informally referred to as Java Collections, and the Generic Collection Library for Java (JGL) by ObjectSpace [66]. They both provide containers and iterators; JGL also provides algorithms and function objects. Similar considerations to those made for STL apply for these two libraries.

3.4 LEDA

LEDA [60, 61], the Library of Efficient Data Structures and Algorithms,[6] 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 [13, 59].

Ease of Use LEDA's data structures and algorithms are well documented with respect to their space and time complexity. Of particular importance for the ease of use of LEDA is the textbook-like look and feel of the code, written following the "literate programming" approach. Every new release is accompanied by a detailed user manual.

Efficiency Most of the implemented algorithms are asymptotically optimal. Altogether, LEDA is moderately efficient; for a particularly complex data structure as the graph, a factor of 4 performance loss is claimed with respect to an ad hoc implementation. LEDA provides its own efficient memory manager. Independent item types (i.e., items that are not part of a collection data type), such as points, lines, and segments, are implemented using smart pointers with reference counting capabilities (see Items 28–29 of Ref. [63]), called handle types (see Chapter 13 of Ref. [61] for details); this guarantees efficient assignment, copy, and destruction operations, and identity testing.

Flexibility Most data types in LEDA are parameterized with respect to the type of their elements by means of the template mechanism. Some limitations[7] of the template mechanism are nicely

[6] http://www.mpi-sb.mpg.de/LEDA/.

[7] Although powerful and elegant, the template mechanism presents also some drawbacks. For each template class instantiation with a different parameter type, the code of the class is duplicated; this increases code size and compilation time. Template classes cannot be precompiled and organized in a library to be later linked to an application; rather, they have to be included in the application and compiled with it. Finally, the template mechanism does not allow collections of heterogeneous elements.

avoided in LEDA: each parameterized data type is realized by a pair of classes, a class for the abstract data type (or interface) and one for the data structure (or implementation). Only the former, whose size is usually quite small, uses the template mechanism (see Chapter 13 of Ref. [61] for details). The same mechanism is also adopted to let the user select one of the multiple possible implementations for an abstract data type. For example, there exist implementations of a priority queue as a Fibonacci heap, a pairing heap, a k-nary heap, a monotonic heap, and a van Emde Boas tree. It is even possible to add a user-written implementation, as long as it conforms to the interface of a priority queue.

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 can be of two types: real point approximations, using Cartesian coordinates, or exact rational points, 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 6.6, there are cases in which this predicate can be evaluated in a more efficient way.

Reliability Multiple types of arithmetic, such as arbitrary length integer numbers, rational numbers, and algebraic real numbers, are available, and allow exact computations with different degrees of efficiency. In particular, algebraic real numbers [13, 15], thanks to the use of an arithmetic filter, are the most powerful and usually more efficient arithmetic type provided by LEDA. Different types of arithmetic cannot currently coexist in geometric data structures.

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. The LEDA Extension Packages framework has been recently added to integrate extensions of the core system.

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 has evolved from an initial flat design to a modular one. Currently, four large modules can be identified: basic data types (including dictionaries and priority queues), graphs and related algorithms, a geometric kernel, and a visualization support module. The geometric kernel [59] is based on a layered design: it is composed by an arithmetic layer, a linear algebra layer, and a geometric layer.

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

Correctness Checking Pioneering work on correctness checkers has been done within LEDA [32, 62].

3.5 CGAL

CGAL and LEDA are separate but complementary initiatives. CGAL [31, 68], the Computational Geometry Algorithms Library,[8] is an initiative involving seven research centers and funded by the European Community. CGAL is still work in progress, but various versions have already been

[8] http://www.cs.uu.nl/CGAL/.

released. It consists of three parts: the kernel, which contains primitive geometric objects, the basic library, which contains basic geometric data structures and algorithms; and the support library, which contains non-geometric support facilities.

Ease of Use A large number of concepts from STL are used in CGAL. This provides a smooth learning curve for new users already familiar with that library. Only a few non-geometric additional concepts are introduced in CGAL, and a new user can start working without knowing all advanced techniques and concepts, such as circulators, creator function objects, or classes for composing function objects. 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 interfacing CGAL with various visualization tools.

Efficiency As in LEDA, implementations of many asymptotically optimal geometric algorithms are provided. Sometimes multiple versions of an algorithm are supplied, where a faster version is obtained, for instance, by ignoring degeneracies. The user is allowed to specify the type of arithmetic used for representing primitive geometric objects, so that computationally expensive exact arithmetic is used only where actually needed. This is obtained through the C++ template mechanism.

Flexibility The current CGAL kernel provides two families of implementations of the basic geometric objects: one based on a representation of points by Cartesian coordinates, and the other based on a representation of points by homogeneous coordinates. The CGAL kernel is also open in terms of type of arithmetic used: by a template parameter, the user can specify the number type of the coordinates. Possible choices include C++ primitive number types (`int`, `long`, `float`, and `double`), LEDA number types (e.g., arbitrary length integer numbers and algebraic real numbers), GNU Multiple Precision Arithmetic Library [42] number types, CGAL interval arithmetic number types, and even user-provided number types.

Of particular importance, in the CGAL implementation of geometric algorithms, is the use of *traits* classes. All types and geometric primitives necessary for a certain algorithm are encapsulated in a traits class, which is then passed as a template argument to the algorithm. The algorithm can thus be implemented independently from the particular representation chosen for the geometric objects and from the computation performed for evaluating the geometric primitives.

Reliability Exact geometric computations can be guaranteed, if needed, by properly choosing the type of representation and the number type for basic geometric objects. Two particularly interesting choices [11], in this respect, are the Cartesian coordinates with LEDA algebraic real numbers, and the homogeneous coordinates with LEDA arbitrary length integer numbers.

Extensibility A certain degree of extensibility is present, as shown by the possibility of importing arithmetic types. Further, the generic programming paradigm adopted in the implementation and the compliance with STL should facilitate the integration of external contributions into the library.

Modularity CGAL is organized into four large modular units: a core library of basic non-geometric functionalities; a geometric kernel containing constant-size geometric objects (such as points, lines, segments, triangles, and tetrahedra) and predicates on them; a basic library of more complex geometric objects, data structures, and algorithms; and a support library. Geometric objects are not part of a hierarchy, but are rather organized in independent packages. The interoperation between the basic library and the geometric kernel is based on the use of the traits classes described above.

Functionality CGAL contains a large collection of 2D and 3D geometric objects and data structures, and various geometric algorithm implementations.

Correctness Checking A sophisticated system of on demand checks and warnings, based on pre- and post-conditions, 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 to prove the applicability of some advanced software design concepts to geometric computing through a vertical case study rather than to describe the design of a comprehensive geometric computing library. Thus, in this section, we review only those aspects of the preliminary design of JDSL/GeomLib that are relevant to the case study presented in Section 6. The JDSL/GeomLib project is described in greater detail in Refs. [36, 39, 40, 41].

Our design relies on various object-oriented design concepts, such as *object*, *class*, *interface*, *inheritance*, *polymorphism*, *dynamic binding*, and *design pattern*. We briefly review them in Appendix A, and refer the interested reader to, e.g., Refs. [16, 35, 80, 84]. 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 factory* design pattern [35] 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 (directly or through inheritance) correspond to different possible views.

The architecture of JDSL/GeomLib 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.[9] 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/GeomLib does not directly depend on Java; its validity extends to C++ and other object-oriented programming 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 present

[9] A Java *package* 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.

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 [41, 65, 70]. The combinatorial component is further divided into two subcomponents, one for the basic data structures and one for the combinatorial graph.

4.1.1 The basic data structures subcomponent

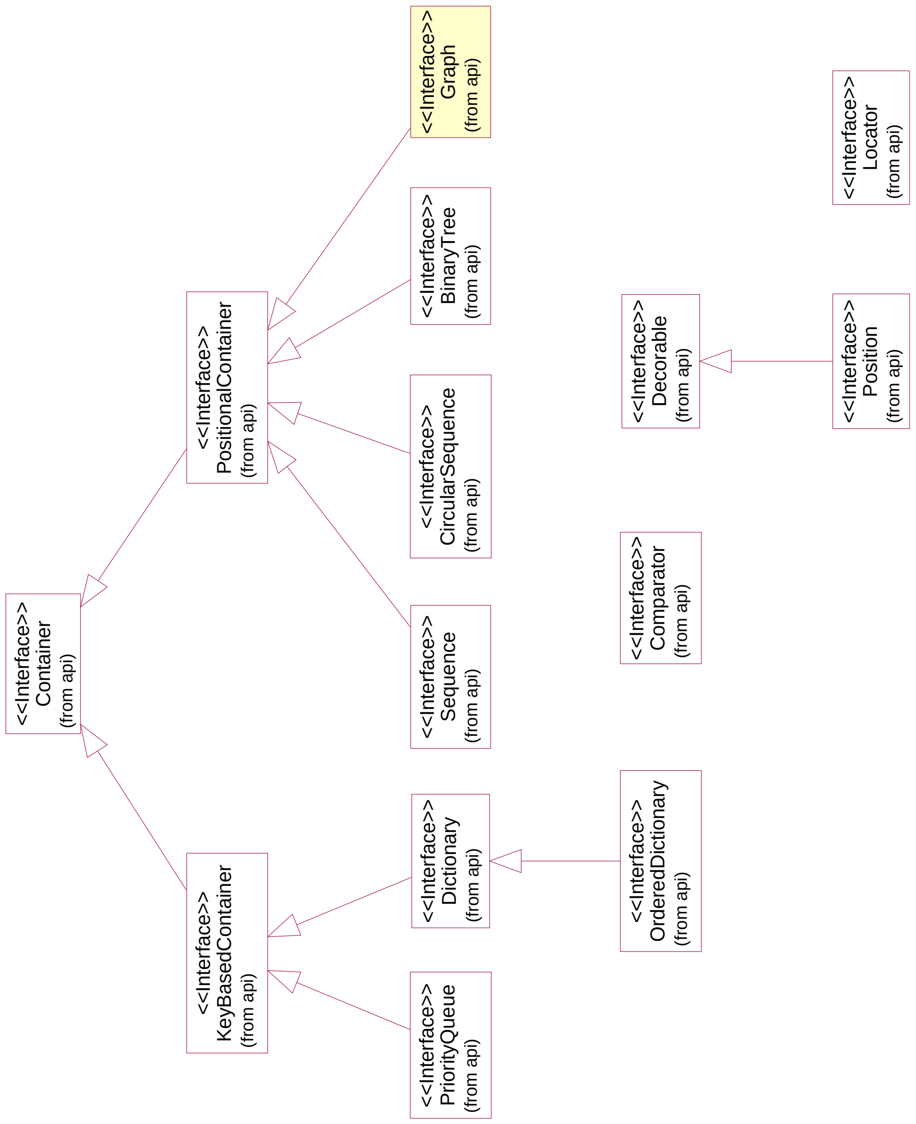
In this subcomponent, most of the basic data structures are defined and implemented. Containers can be roughly divided into two categories: *positional* containers and *key-based* containers. Typical positional containers are *sequences*, *trees*, and *graphs*; typical key-based containers are *priority queues* and *dictionaries*. The common properties of the constituents of a positional container are abstracted in the concept of *position*. 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, called *locator*, is used to access a specific (key,element) pair in a key-based container in constant time; its description, however, is beyond the scope of this paper. A high-level view of the interface hierarchy for the basic data structures subcomponent is shown in Figure 1. The actual implementation contains two parallel interface hierarchies, since each basic data structure interface in Figure 1 corresponds to two interfaces: one for the immutable version of the data structure (whose name is prefixed by `Inspectable`) and the other, extending the first, for the mutable version.

The basic data structures subcomponent corresponds to the `jdsl.core.api`, `jdsl.core.ref`, and `jdsl.core.algo` packages 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 `size()`, returning the number of elements in the container, `elements()`, returning an enumeration of the elements in the container, and `isEmpty()`.

`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`, `BinaryTree`, and `Graph` interfaces. It provides meth-



<<Interface>> Container (from api)
<<Interface>> PositionalContainer (from api)
<<Interface>> KeyBasedContainer (from api)
<<Interface>> Graph (from api)
<<Interface>> BinaryTree (from api)
<<Interface>> CircularSequence (from api)
<<Interface>> Sequence (from api)
<<Interface>> Dictionary (from api)
<<Interface>> PriorityQueue (from api)
<<Interface>> Decorable (from api)
<<Interface>> Comparator (from api)
<<Interface>> OrderedDictionary (from api)
<<Interface>> Locator (from api)
<<Interface>> Position (from api)

Figure 1: A high-level view of the interface hierarchy for the basic data structures subcomponent of JDSL.

ods 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.

`Sequence` This interface describes a sequence and provides methods such as `before (Position)` and `after(Position)`, returning the positions before and after a given position, `first()` and `last()`, returning the first and last positions of the sequence, `insertAfter(Position, Object)`, inserting a new position storing an object after a given position, and `removeAfter (Position)`, removing the position after a given one. An implementation of the `Sequence` interface is used in the preprocessing of the case study.

`BinaryTree` This interface describes a binary tree and provides methods such as `leftChild (Position)` and `rightChild(Position)`, returning the left and right child of a given node, `cut(Position)`, removing the subtree rooted at a given node, and `link(Position, BinaryTree)`, adding a subtree at a given leaf. An implementation of the `BinaryTree` interface is extensively used as a search structure in the case study.

`Decorable` This interface is used to implement the *decorator* design pattern [35]. The motivation of this pattern is to attach additional, named attributes to individual objects rather than to an entire 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 as visited a vertex of a graph, or associating a weight to the nodes of a tree). The interface provides methods such as `set(Object,Object)` and `get(Object)` for setting and getting the value of an attribute, `has(Object)` for testing the existence of an attribute, and `destroy(Object)` for removing it.

JDSL provides various classes implementing the above interfaces, and these classes are often internally reused in the implementation of more advanced data structures. Note, however, that any class implementing one of the above interfaces, either directly or through a wrapper class, could be used in place of the JDSL classes. Alternative implementations of the above interfaces can also be used as auxiliary data structures; for instance, if a sequence is used as an auxiliary data structure in the implementation of an algorithm, class `LinkedList` from the `java.util` package of Sun's Java Development Kit (JDK), or class `Dlist` from the `com.objectspace.jgl` package of ObjectSpace's Generic Collection Library for Java (JGL) can be used as alternatives to JDSL classes `NodeSequence` and `ArraySequence`.

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 as 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 interface hierarchy for the graph subcomponent is shown in Figure 2; its detailed description is beyond the scope of this paper.

The graph subcomponent corresponds to the `jdsl.graph.api`, `jdsl.graph.ref`, and `jdsl.graph.algo` packages of JDSL. Some of the most relevant interfaces for our case study are the following:

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

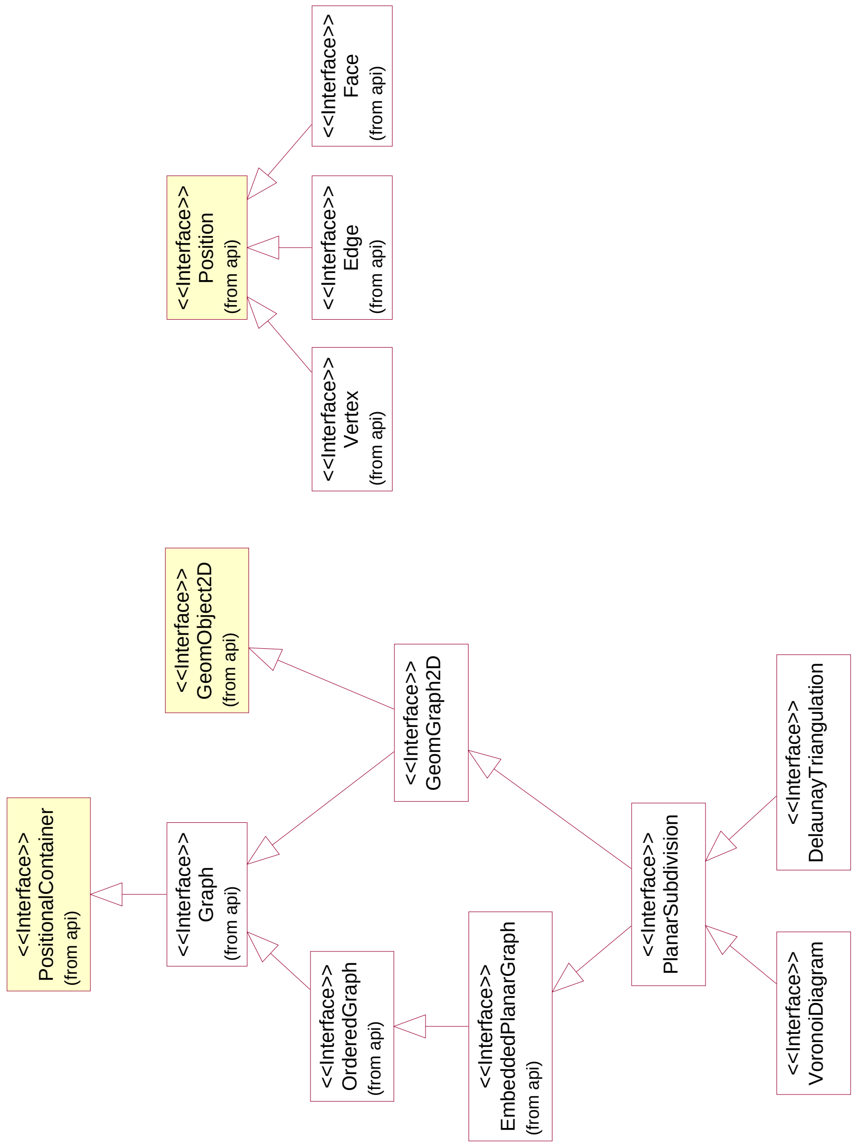


Figure 2: A high-level view of the interface hierarchy for the graph subcomponent, the topological component, and the advanced geometric subcomponent of JDSL.

`Graph` This interface describes a combinatorial graph. It provides methods such as `numVertices()`, returning the number of vertices of the graph, `degree(Vertex)`, returning the degree of a vertex, `incidentEdges(Vertex)`, returning the edges incident with a vertex, `origin(Edge)` and `destination(Edge)`, returning the origin and the destination of a directed edge, and `insertEdge (Vertex,Vertex,Object)` inserting an edge storing an object between two vertices. The methods from this interface are inherited by the `PlanarSubdivision` interface and are used in the preprocessing of the case study.

As an example of internal reuse of code, the existing implementation of the `Graph` interface uses a class implementing 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., as 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. A high-level view of the interface hierarchy for the topological component is shown in Figure 2; its detailed description is beyond the scope of this paper.

The topological component corresponds to the `jdsl.map.api`, `jdsl.map.ref`, and `jdsl.map.algo` packages of JDSL. Some of the most relevant interfaces for our case study 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. The methods from this interface are inherited by the `PlanarSubdivision` interface and are used in the preprocessing of the case study.

`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, `incidentEdges(Face)`, returning the edges incident with a face, `leftFace(Edge)` and `rightFace(Edge)`, returning the face to the left or to the right of a directed edge, and `dual()`, returning the topological dual embedded planar graph. The methods from this interface are inherited by the `PlanarSubdivision` interface and are used in the preprocessing of the case study.

There exist several possible representations for an embedded planar graph, such as the DCEL representation, originally presented in Ref. [64] and later refined (see, e.g., Ref. [21]), the quad-edge representation [44], and the dynamic representations described in Refs. [29, 30, 81]. Each representation presents advantages and disadvantages, and 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., Ref. [83]).

The JDSL implementation of the `EmbeddedPlanarGraph` interface is based on the *incidence graph* representation described in Chapter 11 of Ref. [26]. The embedded planar graph G is represented through a graph G' such that: (i) each vertex of G' corresponds to either a vertex, or an edge, or a face of G; (ii) each vertex of G' corresponding to an edge e of G has four adjacent vertices that correspond 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 forms, with the arithmetic component, 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 and algorithms.

4.3.1 The basic geometric objects subcomponent

In this subcomponent, basic geometric objects, such as points, lines, rays, segments, circles, etc., are defined and implemented. Currently, interfaces have been defined and implementations have been provided only for two-dimensional basic geometric objects. The interface hierarchy for the basic geometric objects subcomponent is shown in Figure 3.

This subcomponent corresponds to the `jdsl.geomobj.api` and `jdsl.geomobj. ref` packages 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 method is `dim()`, returning the dimension of the geometric object. It is extended by the "typing" interfaces `GeomObject2D` and `GeomObject3D`.

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

`LinearCurve2D` This interface describes a linear curve, i.e., (a portion of) a straight line. It extends `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(Point2D,LinearCurve2D)`, testing whether the point is above, on, or below the linear curve, `leftRight(Point2D,LinearCurve2D)`, testing whether the point is to the left, on, or to the right of the linear curve, `aboveBelow(Point2D, Point2D)`, testing whether the first point is above, on, or below the second one, `leftRight (Point2D,Point2D)`, testing whether the first point is to the left, on, or to the right of the second one, `leftRightTurn(Point2D,Point2D,Point2D)`, testing whether the three points form a left turn, a right turn, or are collinear, and `insideOutside(Point2D,Circle2D)`, testing whether the point is inside, on, or outside the circle.

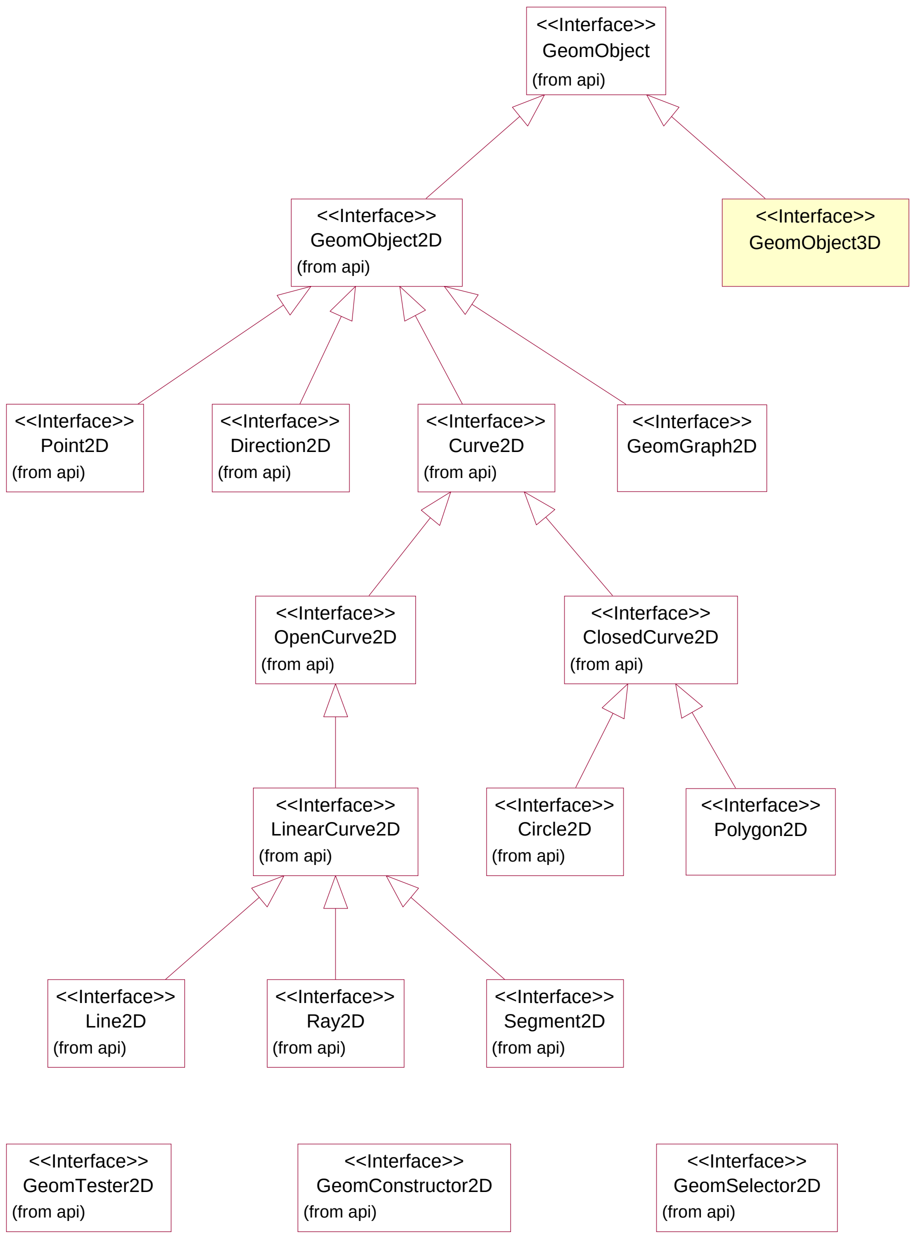


Figure 3: The interface hierarchy for the basic geometric objects subcomponent of JDSL.

For many interfaces of this subcomponent, various implementations are possible, according to which representation is chosen for the basic geometric objects. If we consider a point, for instance, we can represent its coordinates as integer numbers, exact rational numbers, or real number approximations; and if we choose rational numbers, it may be convenient to adopt homogeneous coordinates rather than Cartesian coordinates. Also, it may be useful to have the possibility of choosing other representations for the basic geometric objects, differing not only for the type of arithmetic used, as shown in Section 6.6.

We have grouped all the usual geometric tests and constructions in two interfaces in order to have their implementations localized, thus making their update easier should a new representation of a basic geometric object 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 and constructions should be performed invoking the appropriate methods of the geometric tester and constructor interfaces. 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 modified if such representation changes. Provided that

```
// compute the anchor point and remove it from hull together with all the coincident points
private static void anchorPointSearchAndRemove () {
  // ...
  while (pe.hasMoreElements()) {
    Position pos = (Position)pe.nextElement();
    Point2D p = (Point2D)pos.element();
    int aboveBelow = geomTester.aboveBelow(p,anchorPoint);
    int leftRight = geomTester.leftRight(p,anchorPoint);
    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);
}
```

```
// Graham's scan
private static void scan() {
  // ...
  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));
}
```

Code Fragment 1: A portion of the GeomLib implementation of the Graham's scan algorithm.

the results of the tests and constructions 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 results of the geometric tests and constructions. We will discuss this fundamental issue in Section 4.4.

As a simple example of the above concepts, we present in Code Fragment 1 a portion of the GeomLib implementation of the Graham's scan algorithm for the two-dimensional convex hull problem (see Chapter 15 of Ref. [41] for the complete implementation). The purpose of the two code fragments is to illustrate how, in the auxiliary methods called from the main method, the geometric information of the points is never accessed directly. Objects of type `Point2D` are passed as actual parameters to methods of interface `GeomTester2D`. For instance, method `anchorPointSearchAndRemove()` computes the anchor point, i.e., the bottommost and leftmost point of the set, and removes it from the set together with all its coincident points. To test whether a point is below the current anchor point, or at the same y-coordinate and to its left, methods `aboveBelow(Point2D,Point2D)` and `leftRight(Point2D,Point2D)` of `GeomTester2D` are used. Similarly, in method `scan()`, which scans the sequence of points previously sorted by polar angle, method `leftRightTurn(Point2D,Point2D,Point2D)` of `GeomTester2D` is used to test whether three consecutive points form a left turn. Each of these methods is implemented as a two-step procedure: determine of what classes implementing `Point2D` the parameters are an instance, and execute the code corresponding to that combination of classes. 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 what classes implementing `Point2D` are actually used. In the implementation of the algorithm, all possible inputs are considered, including degenerate ones, such as collinear points or points coincident with the anchor point.

4.3.2 The advanced geometric subcomponent

In this subcomponent, the two-dimensional geometric graph interface and the planar subdivision interface are defined. A two-dimensional geometric graph is a graph whose vertices and edges have some geometric information associated with them, such as a two-dimensional point for each vertex and a two-dimensional curve for each edge. 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 associated with 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 embedded planar graph interface and the two-dimensional geometric graph interface. A high-level view of the interface hierarchy for the advanced geometric subcomponent is shown in Figure 2; its detailed description is beyond the scope of this paper.

This subcomponent corresponds to the `jdsl.geom.api`, `jdsl.geom.ref`, and `jdsl.geom.algo` packages of JDSL. Some of the most relevant interfaces are the following:

`GeomGraph2D` This interface describes a two-dimensional geometric graph and extends both `Graph` and `GeomObject2D`. Its main additional methods are `geomObject2D(Vertex)` and `geomObject2D(Edge)`, returning the two-dimensional geometric object associated with a vertex or an edge, respectively.

`PlanarSubdivision` This interface describes a planar subdivision and extends both `EmbeddedPlanarGraph` and `GeomGraph2D`. Its main additional method is `geomObject2D`

`(Face)`, returning the two-dimensional geometric object associated with a face. An instance of a class implementing the `PlanarSubdivision` interface is passed as input to the preprocessing algorithm of the case study.

The planar subdivision is a good example of the advantages of the interface inheritance mechanism. Through interface multiple inheritance, the same implementation of the planar subdivision presents different views: a simple container, ignoring the combinatorial, topological, and geometric information; a combinatorial or topological graph, ignoring the associated geometric information; a two-dimensional geometric graph, ignoring the topological information; or the combination of a topological graph and a two-dimensional geometric graph. Thus, since the planar subdivision can be viewed as a combinatorial graph, the algorithms defined for a combinatorial graph in the `jdsl.graph.algo` package of JDSL (e.g., the topological sorting algorithm) can be directly applied to the planar subdivision, as done in the preprocessing of the case study.

4.4 The Arithmetic Component

In Section 4.3.1, we have seen how the encapsulation of the geometric information 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 Refs. [3, 14, 24, 28, 33, 34, 43, 47, 48, 49, 57, 76, 78, 85, 86, 87]. The assumption of real number arithmetic has proved unrealistic, since digital computers do not exhibit such capability natively, i.e., in hardware. On the other hand, exact rational arithmetic via software may excessively slow down computations. In light of these problems, the equivalent concepts of *precision* [10], *degree* [57], and *depth of derivation* [86, 87] of a geometric algorithm 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. Namely, a geometric algorithm of degree d requires in its computations a precision that is, in the worst case, about d times that of the input data. 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* [9, 12, 23, 34, 51]. possibly families of filters of progressively increasing power, that, depending upon the values of the test variables, carefully adjust the computational effort, as done for the algebraic real number arithmetic type in LEDA. 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, say, a C++ program should become acceptable thanks to the use of second-generation Java virtual machines or high-performance compilers that produce optimized platform-specific native code (see also Appendix B).

Flexibility In Section 4.3, we have described how the interface inheritance mechanism allows the users of GeomLib to choose the most appropriate representations for the geometric objects of a given problem, without having to modify the geometric program itself. An example of use of this capability is shown in Section 6.6. In addition, GeomLib supports heterogeneous representations for the various geometric objects used in the same program.

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.

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 add new interfaces and 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) to rapidly achieve a high level of functionality, even though this may to some extent reduce efficiency.

Modularity The architecture of JDSL/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.

Functionality Currently, there exists a preliminary prototype implementation of JDSL/GeomLib, consisting of the combinatorial and topological components, and of a subset of the geometric component, including all 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.

Correctness Checking Currently, GeomLib does not 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 show how the *binary space partition search* algorithmic pattern provides a unifying framework for the implementation of two major techniques for planar point location: the *chain method* [27, 55] and the *trapezoid method* [71]. Both methods are very efficient in practice, as reported in Ref. [25]. We provide a concrete example of the use of GeomLib and of the application of algorithm engineering techniques to the implementation of geometric algorithms. We show how the use of the above algorithmic pattern within GeomLib supports the fast prototyping of robust and reliable data structures for point location.

Planar point location is a fundamental search operation in computational geometry, and has been the target of substantial research. Surveys on methods for planar point location are presented in Refs. [72, 77]. 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, we wish to determine which elementary 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.

For concreteness, we will describe the binary space partition search algorithmic pattern through its specialization to the chain method. Then, we will show how the same algorithmic pattern can be specialized to the trapezoid method. Throughout the section, we use the Unified Modeling Language (UML) formalism [8, 50, 74] to represent the conceptual models of the algorithmic pattern.

As for many algorithms in the combinatorial component of JDSL, these two point location algorithms are implemented as algorithm objects, following the *strategy* design pattern [35]. In particular, we have defined an interface `PointLocation`, whose only method is `query(Point2D)`, and two classes, `ChainMethod` and `TrapezoidMethod`, implementing that interface (see Figure 4). An object of type `PlanarSubdivision` (an interface defined in the geometric component of JDSL/GeomLib) is passed as a parameter to the constructors of the two classes, which are responsible for the preprocessing.

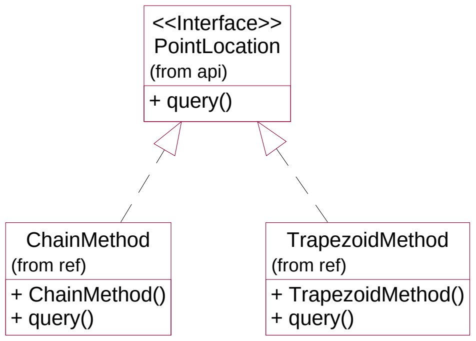


Figure 4: The conceptual model of the point location algorithm interface and classes. Dashed segments connect classes to the interfaces they implement.

6.1 Review of the Chain Method

In this section, we consider the chain method for planar point location by Lee and Preparata [55]. In particular, we focus on its variant described in Chapter 11 of Ref. [26]. We recall that the search structure requires $O(n)$ space, can be constructed in $O(n \log n)$ time, and allows queries in $O(\log^2 n)$ time.

The 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. The separators used to recursively decompose subdivision S are monotone chains of vertices and edges of S, called for brevity *chains*. As described in Section 1.3, the recursive decomposition of S is represented by means of a binary tree T. Each node of T is thus associated with a chain of S, and each leaf node is associated with a face of S. In the preprocessing phase, the chains are chosen so that they leave about half of the regions to the left, and half to the right. Hence, tree T has logarithmic height.

Although an edge of S may belong to several chains, it is stored only once at the highest node of T where it appears, in order to keep the space requirement linear. Hence, chains may have "gaps" corresponding to edges that are not actually stored. In order to deal with "gaps", faces and chains are labeled with consecutive integers, and stored in T so that their labels appear consecutively in the inorder traversal of T.

A query for a point q consists of traversing a downward path in T starting at the root. At each internal node μ, point q is "discriminated" against the chain associated with μ:

1. If q is to the left of the chain, the search continues on the left child of μ.

2. If q is to the right of the chain, the search continues on the right child of μ.

3. If q is on a vertex or edge of the chain, the search terminates and that vertex or edge is returned.

Thus, if q is contained in a face f of S, the search terminates at the leaf of T associated with f, while if q is on a vertex or edge of S, the search terminates at the first internal node whose chain contains q. Also, during the search, the range of the labels of the faces and chains that are candidates for containing q is maintained.

The discrimination of point q against a chain consists of:

1. Determining the component of the chain that is horizontally in front of q; this component is either a vertex, or an edge, or a subchain of horizontal edges.

2. Testing whether q is to the left of the component, to the right of the component, or on (a vertex or edge of) the component. This is done by means of a geometric left/right test if the component is explicitly stored and by using the label of the chain and the current label range if the component is on a "gap" of the chain.

6.2 The Binary Space Partition Search Algorithmic Pattern

The design of the binary space partition search algorithmic pattern is based on the concepts of *search region* and *separator*, characterized by the following properties:

- A search region is recursively decomposed by means of separators.

- In the discrimination step, the separator itself may be a search region in a lower-dimensional space.

The basic operation in the search algorithm is the discrimination of the query point against the separator of the current region, 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 search region are chains, the elementary regions are faces, and the search structure representing the recursive decomposition is called *separator tree*.

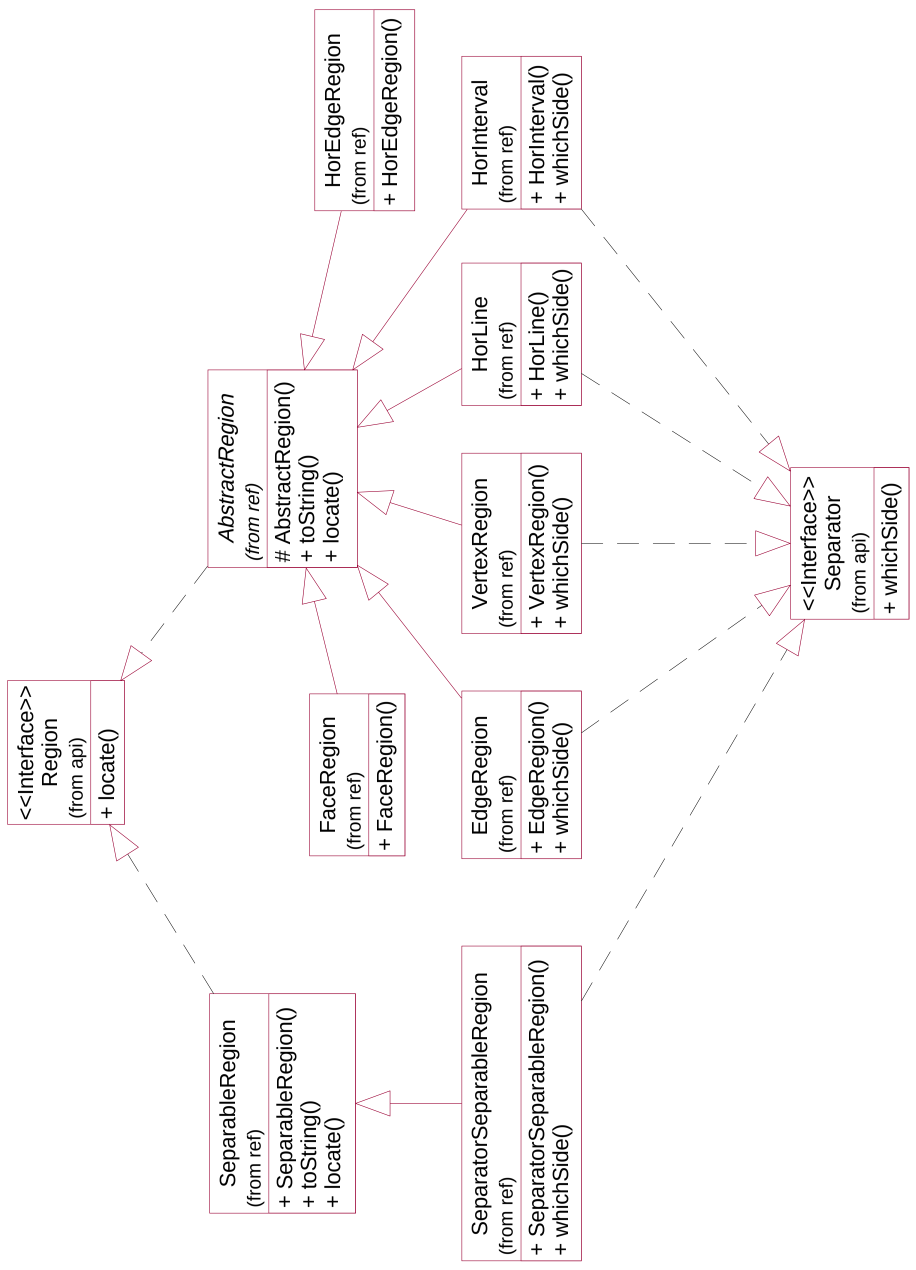


Figure 5: The conceptual model of the binary space partition search algorithmic pattern. Dashed segments connect classes to the interfaces they implement, and solid segments connect classes to the classes they extend.

The separators of a chain search region are horizontal lines, the elementary regions are edges, and the search structure representing the recursive decomposition is called *chain tree*; each horizontal line corresponds to a vertex or a horizontal subchain. The separators of a horizontal subchain search region are vertices, the elementary regions are horizontal edges, and the search structure representing the recursive decomposition is called *horizontal subchain tree.*

Using the above concepts, the search algorithm can be modeled as an algorithmic pattern, called binary space partition search, characterized by the interaction of objects of the following types:

Region The object in which the query point is searched.

Separator The object against which the query point is discriminated. Note that a separator is a region.

Search Status The object that provides the next region to be searched, based on the result of the discrimination of the query point against the separator of the current region.

Note how in the binary space partition search algorithmic pattern, the structural properties of the search regions are kept separate from the data structures used to represent their recursive decomposition. In particular, each data structure is encapsulated in the corresponding search status. This design decision is critical in allowing us to implement different point location methods as specializations of a common algorithmic pattern.

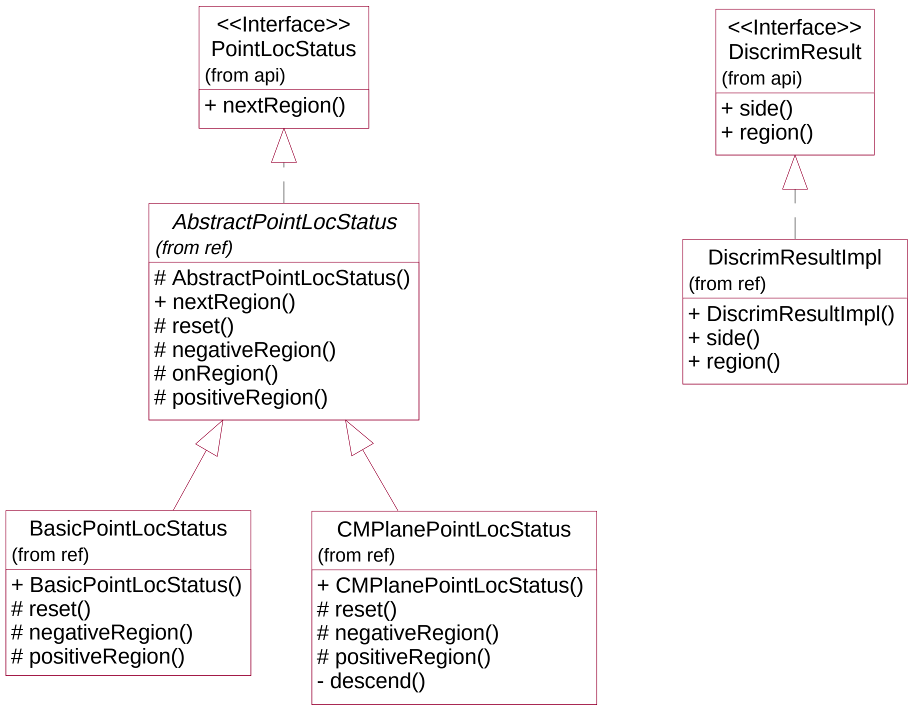


Figure 6: The conceptual model of the binary space partition search algorithmic pattern (continued). Dashed segments connect classes to the interfaces they implement, and solid segments connect classes to the classes they extend.

The conceptual model of the binary space partition search algorithmic pattern is shown in Figures 5 and 6. We have captured the main aspects of the algorithmic pattern in the four following Java interfaces.[10]

`Region` (see Code Fragment 2) This interface describes a region in which the location is performed. Its only method is `locate(Point2D)`, returning the face, edge, or vertex region containing the query point.

`Separator` (see Code Fragment 2) This interface describes the separator of a separable region. Its only method is `whichSide(Point2D)`, returning the result of the discrimination of the query point against this.

`DiscrimResult` (see Code Fragment 2) This interface describes the result of the discrimination of the query point against a separator. Its two methods are `side()`, returning the side of the separator containing the query point, and `region()`, returning the region acting as a separator in the discrimination.

`PointLocStatus` (see Code Fragment 2) This interface describes a search status, which keeps track of the status of the search in a separable region. Its only method is `nextRegion (DiscrimResult)`, returning the region in which to continue the search.

These interfaces are implemented by the following classes, two of which abstract.[11] In this section we present the code for the most relevant ones, while the code for the remaining ones is given in Appendix C.

`SeparableRegion` (see Code Fragment 3) implementing interface `Region`. This class describes a region that is divided by a separator. In particular, it is used to model a recursive decomposition of the plane and of a horizontal subchain. 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 two instance variables `separator_` and `status_` store a reference to the separator and to the search status of the object, respectively.

`SeparatorSeparableRegion` (see Code Fragment 3) extending class `SeparableRegion` and implementing interface `Separator`. It is used to model a recursive decomposition of a chain, and, as such, a separator of the plane. Its only additional method is `whichSide()` of interface `Separator`. Note how this method is implemented similarly to method `locate()` of class `SeparableRegion`.

`AbstractPointLocStatus` (see Code Fragment 4) implementing interface `PointLocStatus`. It is used to maintain the status in the search structure associated with a separable region. Method `nextRegion(DiscrimResult)` is implemented by invoking one of the three methods `negativeRegion(Region)`, `onRegion(Region)`, `positiveRegion(Region)`, based on the result of the discrimination of the query point against the current separator. The first and last of these methods are declared abstract and are implemented by the two subclasses `BasicPointLocStatus` and `CMPlanePointLocStatus`.

[10] A Java *interface* consists of a set of methods declared with their parameter types, return type, and exceptions thrown; it may also contain a set of constants. Interfaces can participate in multiple inheritance, and one of their main uses is for typing purposes.

[11] A Java *abstract class* is a partially implemented class that is extended by other classes; it cannot be instantiated. Its purpose is to factor out some common features of its subclasses to avoid code duplication.

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

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

}
```

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

  /**
   * @param q the query point
   * @return whether q is on the GeomTester2D.NEGATIVE side of this,
   * GeomTester2D.ON this, or on the GeomTester2D.POSITIVE side of this
   */
  public DiscrimResult whichSide (Point2D q);

}
```

```
/**
 * The result of the discrimination of the query point with respect to the separator
 */
public interface DiscrimResult {

  /**
   * @return the side of the separator containing the query point
   */
  public int side ();

  /**
   * @return the region acting as a separator in the discrimination
   */
  public Region region ();

}
```

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

  /**
   * @param dr the result of the discrimination of the query point against the current separator
   * @return the next region according to the result of the discrimination of the query point
   * against the current separator
   */
  public Region nextRegion (DiscrimResult dr);

}
```

Code Fragment 2: Interfaces `Region`, `Separator`, `DiscrimResult`, and `PointLocStatus`.

```
public class SeparableRegion implements Region {

  // instance variable(s)

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

  // constructor(s)

  public SeparableRegion (Separator separator, PointLocStatus status) {
    separator_ = separator;
    status_ = status;
    label_ = "";
  }

  // instance method(s) from Region

  public Region locate (Point2D q) {
    Region r = status_.nextRegion(separator_.whichSide(q));
    return r.locate(q);
  }

  // instance method(s) from java.lang.Object

  public String toString () {
    return label_;
  }

}
```

```
public class SeparatorSeparableRegion extends SeparableRegion implements Separator {

  // constructor(s)

  public SeparatorSeparableRegion (Separator separator, PointLocStatus status) {
    super(separator,status);
  }

  // instance method(s)

  public DiscrimResult whichSide (Point2D q) {
    Separator s = (Separator)status_.nextRegion(separator_.whichSide(q));
    return s.whichSide(q);
  }

}
```

Code Fragment 3: Classes `SeparableRegion` and `SeparatorSeparableRegion`.

`BasicPointLocStatus` (see Code Fragment 5) extending class `AbstractPointLocStatus`. It is used to maintain the status in the search structure associated with a chain or a horizontal subchain. Note that the descent along the search tree takes place in methods `negativeRegion (Region)` and `positiveRegion (Region)`.

`CMPlanePointLocStatus` (see Code Fragment 6) extending class `AbstractPointLocStatus`. It is used to maintain the status in the search structure associated with the plane. We recall that it may not be necessary to discriminate the query point against some chains, namely those that have a "gap" at the ordinate of the query point (see Chapter 11 of Ref. [26] for details). It follows that the descent mechanism in the separator tree is different from that

```
public abstract class AbstractPointLocStatus implements PointLocStatus {

  // instance variable(s)

  protected BinaryTree tree_;
  protected Position current_;

  // constructor(s)

  protected AbstractPointLocStatus (BinaryTree tree) {
    tree_ = tree;
  }

  // instance method(s)

  public Region nextRegion (DiscrimResult dr) {
    Region region;
    switch (dr.side()) {
    case GeomTester2D.NEGATIVE:
      region = negativeRegion(dr.region());
      if (tree_.isExternal(current_))
        reset();
      break;
    case GeomTester2D.ON:
      region = onRegion(dr.region());
      reset();
      break;
    case GeomTester2D.POSITIVE:
    default:
      region = positiveRegion(dr.region());
      if (tree_.isExternal(current_))
        reset();
    }
    return region;
  }

  protected abstract void reset ();

  /**
   * @return the next region when the result of the discrimination is GeomTester2D.NEGATIVE
   */
  protected abstract Region negativeRegion (Region r);

  /**
   * @return the next region when the result of the discrimination is GeomTester2D.ON
   */
  protected Region onRegion (Region r) {
    return r;
  }

  /**
   * @return the next region when the result of the discrimination is GeomTester2D.POSITIVE
   */
  protected abstract Region positiveRegion (Region r);

}
```

Code Fragment 4: Abstract class `AbstractPointLocStatus`.

```
public class BasicPointLocStatus extends AbstractPointLocStatus {

  // constructor(s)

  public BasicPointLocStatus (BinaryTree tree) {
    super(tree);
    reset();
  }

  // instance method(s)

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

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

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

}
```

Code Fragment 5: Class `BasicPointLocStatus`.

in the chain tree and in the horizontal subchain tree; hence, the different implementation of methods `negativeRegion(Region)` and `positiveRegion(Region)`.

`AbstractRegion` (see Code Fragment 7) extending class `HashtableDecorable` and implementing interface `Region`. Class `HashtableDecorable` is defined in the `jdsl.core.ref` package of JDSL and implements the `Decorable` interface (see Section 4.1.1); thus, all the subclasses of `AbstractRegion` can be decorated.

`FaceRegion` (see Code Fragment 7) extending class `AbstractRegion`. It models a face of the planar subdivision.

`HorEdgeRegion` (see Code Fragment 7) extending class `AbstractRegion`. It models a horizontal edge of the planar subdivision. At the same time, it acts as one of the components of a horizontal subchain.

`EdgeRegion` (see Code Fragment 8) extending class `AbstractRegion` and implementing interface `Separator`. It models an edge of the planar subdivision. At the same time, it acts as one of the components of a chain.

`VertexRegion` (see Code Fragment 8) extending class `AbstractRegion` and implementing interface `Separator`. It models a vertex of the planar subdivision. At the same time, it acts as one of the components of a chain, and as a separator for a horizontal subchain.

`HorLine` (see Code Fragment 9) extending class `AbstractRegion` and implementing interface `Separator`. It models a horizontal line passing through a vertex or through a horizontal subchain of the planar subdivision. At the same time, it acts as a separator for a chain. If the query point is on the horizontal line, the region returned as part of the discrimination result is either a

```
public class CMPlanePointLocStatus extends AbstractPointLocStatus {

  // instance variable(s)

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

  // constructor(s)

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

  // instance method(s)

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

  protected Region negativeRegion (Region r) {
    Object lfn = ((AbstractRegion)r).get(LEFT_FACE_TSN_);
    maxFaceNum_ = ((Integer)lfn).intValue();
    descend();
    return (Region)current_.element();
  }

  protected Region positiveRegion (Region r) {
    Object rfn = ((AbstractRegion)r).get(RIGHT_FACE_TSN_);
    minFaceNum_ = ((Integer)rfn).intValue();
    descend();
    return (Region)current_.element();
  }

  private void descend () {
    while (tree_.isInternal(current_)) {
      int cn = ((Integer)current_.get(CHAIN_NUMBER_)).intValue();
      if (cn > maxFaceNum_)
        current_ = tree_.leftChild(current_);
      else if (cn <= minFaceNum_)
        current_ = tree_.rightChild(current_);
      else
        break;
    }
  }

}
```

Code Fragment 6: Class `CMPlanePointLocStatus`.

`VertexRegion`, modeling the corresponding vertex, or a `HorInterval`, modeling the span of a horizontal subchain.

`DiscrimResultImpl` (see Code Fragment 9) implementing interface `DiscrimResult`.

`HorInterval` (see Code Fragment 10) extending class `AbstractRegion` and implementing interface `Separator`. It models the (possibly open) horizontal interval spanned by a horizontal subchain of the planar subdivision. At the same time, it acts as one of the components of a chain. If the query point is within the interval, two cases are possible: (i) the query point is coincident with one of the two endpoints of the interval; then the region returned as part of the discrimination result is a `VertexRegion`, modeling the corresponding vertex; (ii) otherwise, the region returned as part of the discrimination result is a `SeparableRegion`, modeling a recursive decomposition of the corresponding horizontal subchain.

We now describe in some detail how the various regions are arranged in the search structures. In the implementation of the separator tree, of the chain trees, and of the horizontal subchain trees, we make use of class `NodeBinaryTree`, implementing interface `BinaryTree`, from the combinatorial component of JDSL.

- For the recursive decomposition of each horizontal subchain, we use a `NodeBinaryTree` object to represent the corresponding horizontal subchain tree. Each leaf of the tree corresponds to either an edge or to a "gap" of the horizontal subchain; in the former case it stores a `HorEdgeRegion` object, in the latter the value `null`. Each internal node of the tree stores a `SeparableRegion` object. The separator of this `SeparableRegion` is a `VertexRegion` object, referenced by the instance variable `separator_`. The status of a horizontal subchain tree search structure is maintained by a `BasicPointLocStatus` object, referenced by the instance variable `status_` of the `SeparableRegion` objects stored in the internal nodes of the tree.

- Each `HorInterval` object stores in its instance variable `onRegion_` a reference to the `SeparableRegion` object stored in the root of the `NodeBinaryTree` representing the corresponding horizontal subchain tree.

- For the recursive decomposition of each chain, we use a `NodeBinaryTree` object to represent the corresponding chain tree. Each leaf of the tree may correspond to an edge or to a "gap" of the chain; in the former case it stores an `EdgeRegion` object, in the latter the value `null`. Each internal node of the tree stores a `SeparatorSeparableRegion` object. The separator of this `SeparatorSeparableRegion` is a `HorLine` object, referenced by the instance variable `separator_`. The status of a chain tree search structure is maintained by a `BasicPointLocStatus` object, referenced by the instance variable `status_` of the `SeparatorSeparableRegion` objects stored in the internal nodes of the tree.

- Each `HorLine` stores in its instance variable `onRegion_` a reference to the region used as a separator for the (horizontal) discrimination of the query point against the chain, if the query point is on the `HorLine`. This region is either a `VertexRegion` or a `HorInterval`.

- For the recursive decomposition of the plane, we use a `NodeBinaryTree` object to represent the corresponding separator tree. Each leaf of the tree stores a `FaceRegion` object, while each internal node stores a `SeparableRegion` object. The separator of this `SeparableRegion` is a `SeparatorSeparableRegion` object, referenced by the instance variable `separator_`; more precisely, it is the `SeparatorSeparableRegion` object stored in the root of the NodeBinaryTree representing the corresponding chain tree. The status of the separator tree search structure is maintained by a CMPlanePointLocStatus object, referenced by the instance variable status_ of the SeparableRegion objects stored in the internal nodes of the tree.

6.3 Interaction between Components

In this section, we describe how the various components of the binary space partition search interact with one another. The key methods of the point location query algorithm are the recursive method locate(Point2D) of class SeparableRegion and method nextRegion(DiscrimResult) of class AbstractPointLocStatus. It is through their interplay that the location of the elementary region containing the query point proceeds.

The process begins by invoking method locate(Point2D) on the SeparableRegion stored in the root of the separator tree. The query point is discriminated against the separator by invoking method whichSide(Point2D) on the object referenced by variable separator_ (we will see the details of the implementation of this method later). From the above described arrangement of the various regions in the search structures, we know that this object is the SeparatorSeparableRegion stored in the root of a chain tree. The result of the discrimination is passed as a parameter when method nextRegion(DiscrimResult) is invoked on the CMPlanePointLocStatus object associated with the separator tree and referenced by variable status_. The region returned by this method depends on the position of the query point with respect to the separator. If the point is on the positive or negative side of the separator, then we descend along the separator tree, and the

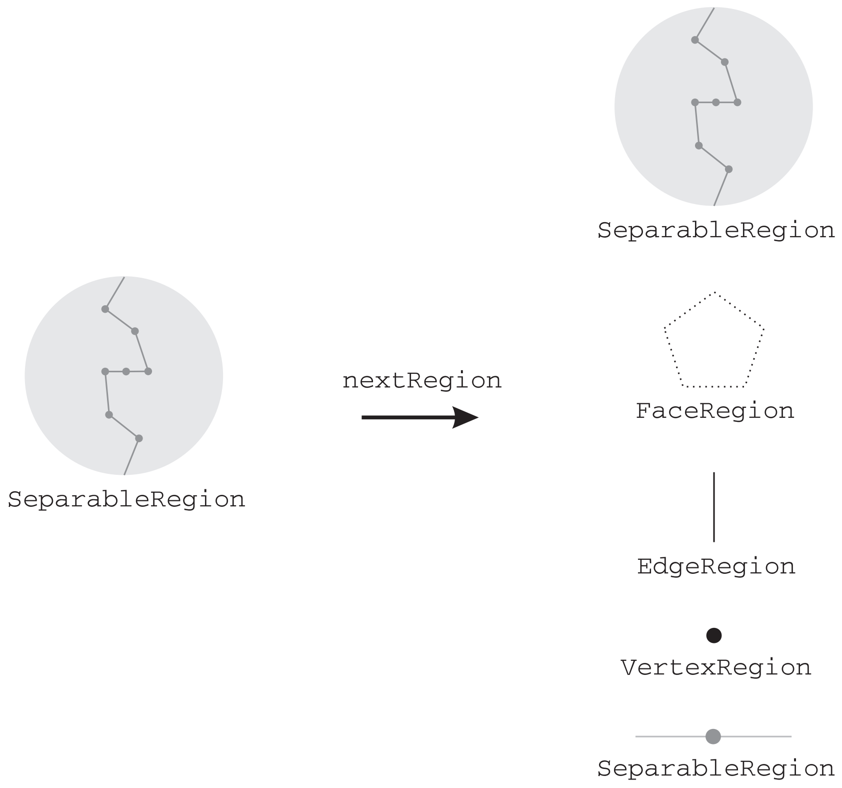


Figure 7: The possible objects returned by method nextRegion(DiscrimResult) when invoked by a SeparableRegion modeling the recursive decomposition of the plane.

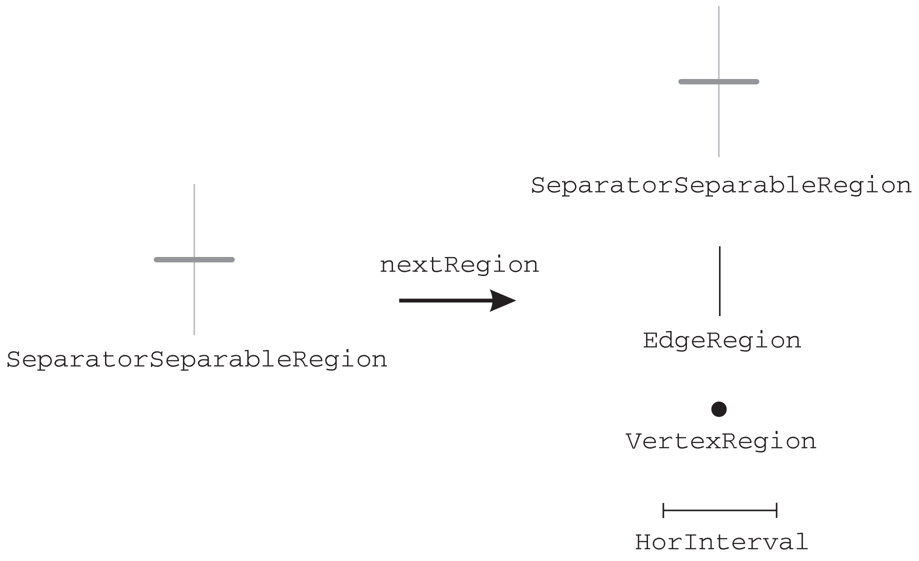


Figure 8: The possible objects returned by method `nextRegion(DiscrimResult)` when invoked by a `SeparatorSeparableRegion` modeling the recursive decomposition of a chain.

region returned is either a new `SeparableRegion` or a `FaceRegion` (if we reach a leaf of the tree). Otherwise, the query point is on the separator, and the region returned is either an `EdgeRegion`, or a `VertexRegion`, or a `SeparableRegion` modeling the recursive decomposition of a horizontal subchain. See the schematic representation in Figure 7. In both cases, method `locate(Point2D)` is recursively invoked on the returned region.

The implementation of method `whichSide(Point2D)` of class `SeparatorSeparableRegion` is similar to that of method `locate(Point2D)` of class `SeparableRegion`. When this method is invoked on the `SeparatorSeparableRegion` stored in the root of a chain tree, the object referenced by variable `separator_` is a `HorLine`. The result of the discrimination is passed as a parameter when method `nextRegion(DiscrimResult)` is invoked on the `BasicPointLocStatus` object associated with the chain tree and referenced by variable `status_`. If the point is on the positive or negative side of the separator, then we descend along the chain tree, and the region returned is either a new `SeparatorSeparableRegion` or an `EdgeRegion` (if we reach a leaf of the tree). Otherwise, the query point is on the separator, and the region returned is either a `VertexRegion` or a `HorInterval`. See the schematic representation in Figure 8. In both cases, method `whichSide(Point2D)` is recursively invoked on the returned separator.

We have seen above that the location process may advance through the invocation of method `locate(Point2D)` on the `SeparableRegion` stored in the root of a horizontal subchain tree. Differently from the `SeparableRegion` objects stored in the nodes of the separator tree, in this case the object referenced by variable `separator_` is a `VertexRegion`. The result of the discrimination is passed as a parameter when method `nextRegion(DiscrimResult)` is invoked on the `BasicPointLocStatus` object associated with the horizontal subchain tree and referenced by variable `status_`. If the point is on the positive or negative side of the separator, then we descend along the horizontal subchain tree, and the region returned is either a new `SeparableRegion` or a `HorEdgeRegion` (if we reach a leaf of the tree). Otherwise, the query point is on the separator, and the region returned is a `VertexRegion`. See the schematic representation in Figure 9. In both cases, method `locate(Point2D)` is recursively invoked on the returned region.

What differentiates classes `BasicPointLocStatus` and `CMPlanePointLocStatus` are methods

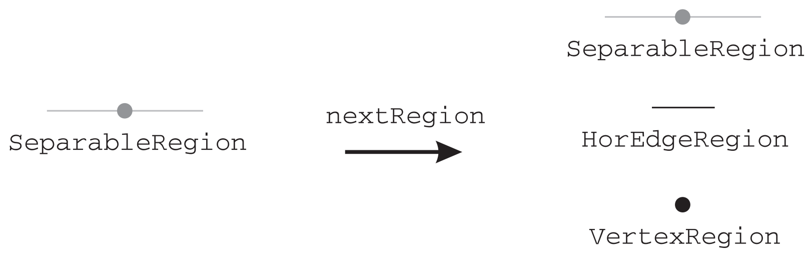


Figure 9: The possible objects returned by method `nextRegion(DiscrimResult)` when invoked by a `SeparableRegion` modeling the recursive decomposition of a horizontal subchain.

`negativeRegion(Region)` and `positiveRegion(Region)`. While their implementation in class `BasicPointLocStatus` is quite intuitive, their implementation in class `CMPlanePointLocStatus` requires some comments. In the preprocessing of the chain method (see Chapter 11 of Ref. [26] for details), the l faces of the planar subdivision are numbered from 0 to $l-1$, according to a topological sorting of the vertices of the directed dual planar subdivision. The chains are numbered from 1 to $l-1$, and chain i satisfies the following property: faces 0 to $i-1$ are to its left, and faces i to $l-1$ are to its right. During a query, the interval I of faces possibly containing the query point narrows down after each discrimination of the query point against a chain. In particular, we have seen that the discrimination of the query point against a chain eventually results in the discrimination of the query point against a component s (either an edge, or a vertex, or a horizontal interval) of the chain. If the query point is to the left of s, then the upper value of I is set to the number of the face to the left of s in the planar subdivision. Similarly, if the query point is to the right of s, then the lower value of I is set to the number of the face to the right of s in the planar subdivision. Since discriminating the query point against chains whose number is outside I does not provide any additional information, we can suitably descend along the separator tree until we reach either a leaf or a node whose corresponding chain number is within I. (Note that chains whose number is outside I would present a "gap" at the ordinate of the query point.)

To this purpose, in the preprocessing we decorate each node of the separator tree with the number of the corresponding chain, and each `EdgeRegion`, `VertexRegion` and `HorInterval` with the number of its left and right face in the planar subdivision. The keys used for these decorations are passed as parameters to the constructor of class `CMPlanePointLocStatus`, and stored in variables `CHAIN_NUMBER_`, `LEFT_FACE_TSN_`, and `RIGHT_FACE_TSN_`, respectively. The lower and upper values of interval I are stored in variables `minFaceNum_` and `maxFaceNum_`, respectively. The descent along the separator tree avoiding unnecessary discriminations is implemented in method `descend()`.

As for method `reset()`, implemented by `BasicPointLocStatus` and `CMPlanePointLocStatus`, its purpose is to reset the status of the search structure for a new query, when the current query is completed.

Note how the only classes in the design that interact with the geometric component of JDSL/GeomLib are `VertexRegion`, `EdgeRegion`, `HorLine`, and `HorInterval`. In particular, method `whichSide(Point2D)` from interface `Separator` is the only one that accesses the geometry of the planar subdivision. For an object of class `HorLine`, the associated basic geometric object against which the query point is discriminated is of type `Point2D`, and the (vertical) discrimination is performed through method `aboveBelow(Point2D,Point2D)` of interface `GeomTester2D`. For an object of class `EdgeRegion`, the associated basic geometric object is of type `LinearCurve2D`, and the (horizontal) discrimination is performed through method `leftRight(Point2D, LinearCurve2D)` of interface `GeomTester2D`. For an object of class `VertexRegion` or `HorInterval`, the associated basic geometric objects are of type `Point2D`, and the (horizontal) discrimination is performed through method `leftRight(Point2D, Point2D)` of interface `GeomTester2D`.

6.4 The Trapezoid Method

In this section, we review another planar point location algorithm, the trapezoid method of Preparata [71], and show how it can be obtained by a simple specialization of 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 queries in $O(\log n)$ time.

In the trapezoid method, the search region is recursively decomposed into simpler regions, called *trapezoids*, by means of two types of separators: horizontal lines and edges of the planar subdivision. Again, the basic operation of 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, as well. This method can be implemented using a subset of the classes described in Section 6.2, namely `SeparableRegion`, `BasicPointLocStatus`, `FaceRegion`, `EdgeRegion`, `VertexRegion`, `HorLine`, and `DiscrimResultImpl`. While in the chain method the separator of a `SeparableRegion` is either a `SeparatorSeparableRegion` or a `VertexRegion`, in the trapezoid method, it is either a `HorLine`, or an `EdgeRegion`, or a `VertexRegion`. Thanks to the use of interfaces as object types, and since all these classes implement the `Separator` interface, the implementation of the methods of `SeparableRegion` need not be changed.

In principle, one could even consider a mixed recursive decomposition of the search region: up to a certain level, it could be decomposed, say, using chains, as in the chain method; then, it could be decomposed using horizontal lines and edges, as in the trapezoid method. Note, however, that, even though the query algorithms for the chain method and the trapezoid method can both be modeled through the same algorithmic pattern, the preprocessing algorithms are considerably different, and thus have separate implementations.

6.5 Optimal Chain Method

The chain method has been refined to optimality by Edelsbrunner, Guibas, and Stolfi [27], using fractional cascading [18, 19]. We recall that, in the optimal chain method, the search structure requires $O(n)$ space, can be constructed in $O(n)$ time, and allows queries in $O(\log n)$ time.

We can implement also the optimal chain method within the binary space partition search algorithmic pattern described in the previous sections. The main modification required is the implementation of a new class `FCPointLocStatus` implementing interface `PointLocStatus`. Its (only) instance will maintain the status of the fractional cascading search structure associated with the chains, called *layered DAG* in Ref. [27]. Thus, it will play for the optimal chain method the same role played for the chain method by the (many) `BasicPointLocStatus` objects maintaining the status of the chain trees.

We are currently experimenting with two alternative designs for `FCPointLocStatus`. In the first design, `FCPointLocStatus` stores a single instance of class `IncidenceListGraph`, implementing interface `Graph`. In the second design, it stores a suitably linked collection of instances of a class implementing interface `OrderedDictionary`. All the aforementioned interfaces and classes are part of the combinatorial component of JDSL.

6.6 Test Primitives in Voronoi Diagrams

In Section 6.1, we have shown that, for the binary space partition search algorithmic pattern, basic geometric objects are accessed only through methods of interface `GeomTester2D`, namely `aboveBelow(Point2D,Point2D)`, `leftRight(Point2D,Point2D)`, and `leftRight(Point2D, LinearCurve2D)`. In this section, we show how this design choice ensures high flexibility. In particular, we focus our attention on method `leftRight(Point2D,LinearCurve2D)` when the planar subdivision is a Voronoi diagram. First, we briefly review the technique presented in Ref. [57] for performing proximity queries with optimal degree (a measure of the worst-case arithmetic precision requirement, see Section 4.4), and then show how this technique is fully supported by the architecture of GeomLib.

We assume that the coordinates of the input sites for the construction of the Voronoi diagram are b-bit integers. In a straightforward implementation of a point location algorithm, the vertices of the Voronoi diagram are computed and explicitly stored as rational numbers with homogeneous coordinates (x, y, w), where x and y are $3b$-bit integers, and w is a $2b$-bit integer. As shown in Ref. [57], this implies that the discrimination of the query point against an edge (method `leftRight(Point2D,LinearCurve2D)` of interface `GeomTester2D`) has degree 6.

The alternative approach proposed in Ref. [57] uses a different representation of the diagram, called *implicit* Voronoi diagram, which 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 for each non-horizontal edge e of the diagram, the two sites $l(e)$ and $r(e)$ such that the perpendicular bisector of $l(e)$ and $r(e)$ is the underlying line of 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 the comparison of the (squares of the) Euclidean distances of the query point from sites $l(e)$ and $r(e)$, which 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. In this case, for instance, the explicit and implicit representations of a `LinearCurve2D` are interchangeable, with no need for modifications in the binary space partition search algorithmic pattern. The only requirement is that, in the implementation of interface `GeomTester2D`, each method with a `LinearCurve2D` parameter, e.g., method `leftRight(Point2D,LinearCurve2D)`, have two different implementations, one for each representation.

7 Conclusions and Future Developments

In the first part of the paper, we have overviewed the preliminary design of the GeomLib library of geometric data structures and algorithms in Java, and we have compared GeomLib with other libraries for geometric computing (especially LEDA and CGAL). The design of GeomLib incorporates modern object-oriented programming concepts and provides a framework for rapid prototyping. A key aspect is its hierarchy of interfaces which allows multiple views of the same object. For example, any combinatorial algorithm for graphs and any topological algorithm for embeddings can be executed directly (without adaptation) on a planar subdivision.

In the second part of the paper, we have presented a case study on the prototype implementation of planar point location data structures using GeomLib. Namely, we have shown how the binary space partition search scheme can be developed into an algorithmic pattern by implementing a reusable software component for planar point location, and we have demonstrated how two fundamental point location techniques, the chain method and the trapezoid method, can be obtained by a simple specialization of this component.

To our knowledge, GeomLib is the only library for geometric computing being developed in Java. Preliminary usage of GeomLib as a rapid prototyping tool has been successful. In addition to the case study presented in this paper, GeomLib has been used as a research tool to quickly develop robust and reusable components for graph drawing applications [37]. Also, thanks to GeomLib, students in the computational geometry course at Brown University have been able to complete assignments and projects that are more sophisticated than those done in previous years. On the negative side, the implementation of GeomLib is still at a preliminary stage in comparison with the C++ libraries LEDA and CGAL. Also, the use of Java and the emphasis on generic data structures and on their internal reuse penalizes the runtime of GeomLib components.

Short-term future 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.
- The implementation of the optimal chain method [27] based on fractional cascading. A preliminary design of this implementation is sketched in Section 6.5.
- The implementation of a checker for verifying the consistency of planar subdivisions based on the method given in Ref. [22] and of a framework for checking planar point location queries.
- Experimental testing of planar point location methods has already been performed in the past [25]. We plan to use the prototype described in this paper to perform an experimental test of point location queries in Voronoi diagrams, comparing the standard explicit representation of the diagrams with the implicit representation described in Ref. [57] (see also Section 6.6).

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

- The design and implementation of other algorithmic patterns for geometric computing, e.g., plane-sweep, space-sweep, and randomized incremental construction.
- The implementation of a system of arithmetic filters in the arithmetic component of GeomLib.
- 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 development of applets for the animation of geometric algorithms (see Ref. [45] for an introduction to this topic).
- An investigation of the applicability of the binary space partition search algorithmic pattern to geographical information systems.

Acknowledgments

We would like to thank Franco Preparata and Mike Goodrich for their encouragement and support, and Jim Baker for his collaboration in writing a preliminary version of this paper. We gratefully acknowledge useful discussions with many members of the JDSL/GeomLib team: Ryan Baker, Michael Boilen, Lubomir Bourdev, Ulrik Brandes, Jitchaya Buranahirun, Robert Cohen, David

Emory, Ming En Cho, Natasha Gelfand, Mark Handy, Benoit Hudson, David Jackson, John Kloss, Masi Oka, Lucy Perry, Maurizio Pizzonia, Keith Schmidt, Andrew Schwerin, Galina Shubina, Marco da Silva, and Amit Sobti.

We would also like to thank the anonymous referees for their helpful comments and suggestions on how to simplify the implementation of the case study and improve the presentation.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 2nd edition, 1997. 4, 45

[2] J. Arvo, editor. *Graphics Gems II*. Academic Press, Boston, MA, 1991. 5

[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. 2, 20

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

[5] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *J. Comput. System Sci.*, 47(3):549–595, 1993. 5

[6] J.-D. Boissonnat and F. P. Preparata. Robust plane sweep for intersecting segments. *SIAM J. Comput.*, 29(5):1401–1421, 2000. 2

[7] J.-D. Boissonnat and J. Snoeyink. Efficient algorithms for line and curve segment intersection using restricted predicates. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 370–379, 1999. 2

[8] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading, MA, 1998. 22

[9] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 165–174, 1998. 20

[10] C. Burnikel. *Exact Computation of Voronoi Diagrams and Line Segment Intersections*. Ph.D. thesis, Universität des Saarlandes, Mar. 1996. 4, 20

[11] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Efficient exact geometric computation made easy. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 341–350, 1999. 9

[12] C. Burnikel, S. Funke, and M. Seel. Exact geometric predicates using cascaded computation. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 175–183, 1998. 20

[13] 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. 1, 7, 8

[14] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 16–23, 1994. 2, 20

[15] C. Burnikel, K. Mehlhorn, and S. Schirra. The LEDA class real number. Research Report MPI-I-96-1-001, Max–Planck–Institut für Informatik, Saarbrücken, Germany, Jan. 1996. http://data.mpi-sb.mpg.de/internet/reports.nsf/NumberView/1996-1-001. 8

[16] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985. 10, 43, 44

[17] B. Chazelle et al. Application challenges to computational geometry: CG impact task force report. Technical Report TR-521-96, Princeton Univ., Apr. 1996. http://www.cs.princeton.edu/~chazelle/pubs/CGreport.ps. 1

[18] B. Chazelle and L. J. Guibas. Fractional cascading I. A data structuring technique. *Algorithmica*, 1(3):133–162, 1986. 36

[19] B. Chazelle and L. J. Guibas. Fractional cascading II. Applications. *Algorithmica*, 1(3):163–191, 1986. 36

[20] F. d'Amore, P. G. Franciosa, and G. Liotta. A robust region approach to the computation of geometric graphs. In G. Bilardi, G. F. Italiano, A. Pietracaprina, and G. Pucci, editors, *Algorithms – ESA '98*, volume 1461 of *Lecture Notes Comput. Sci.*, pages 175–186. Springer-Verlag, 1998. 2

[21] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 1997. 15

[22] O. Devillers, G. Liotta, F. P. Preparata, and R. Tamassia. Checking the convexity of polytopes and the planarity of subdivisions. *Comput. Geom. Theory Appl.*, 11(3–4):187–208, 1998. 5, 38

[23] O. Devillers and F. Preparata. A probabilistic analysis of the power of arithmetic filters. *Discrete Comput. Geom.*, 20(4):523–547, 1998. 20

[24] D. P. Dobkin and D. Silver. Applied computational geometry: Towards robust solutions of basic problems. *J. Comput. Syst. Sci.*, 40(1):70–87, 1989. 2, 20

[25] 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. 3, 21, 38

[26] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1987. 2, 16, 22, 28, 35

[27] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15(2):317–340, 1986. 3, 21, 36, 38

[28] 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. 2, 20

[29] 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. 15

[30] 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. 15

[31] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL a computational geometry algorithms library. *Softw. – Pract. Exp.*, 30(11):1167–1202, 2000. 1, 4, 8

[32] U. Finkler, K. Mehlhorn, and S. Näher. An application of partitions: Checking priority queues. In *LEDA: A Platform for Combinatorial and Geometric Computing*, chapter 5.5.3. Cambridge University Press, Cambridge, England, 1999. 5, 8

[33] S. Fortune. Numerical stability of algorithms for 2D Delaunay triangulations. *Internat. J. Comput. Geom. Appl.*, 5(1&2):193–213, 1995. 2, 20

[34] 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. 2, 20

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995. 1, 2, 10, 13, 22, 43, 45

[36] N. Gelfand, M. T. Goodrich, and R. Tamassia. Teaching data structure design patterns. In *Proc. 29th ACM SIGCSE Tech. Sympos.*, pages 331–335, 1998. 1, 10

[37] N. Gelfand and R. Tamassia. Algorithmic patterns for orthogonal graph drawing. In S. H. Whitesides, editor, *Graph Drawing (Proc. GD '98)*, volume 1547 of *Lecture Notes Comput. Sci.*, pages 138–152. Springer-Verlag, 1998. 38

[38] A. S. Glassner, editor. *Graphics Gems*. Academic Press, Boston, MA, 1990. 5

[39] M. T. Goodrich, M. Handy, B. Hudson, and R. Tamassia. Abstracting positional information in data structures: Locators and positions in JDSL. In *OOPSLA '98 Technical Notes*, 1998. 1, 10

[40] M. T. Goodrich, M. Handy, B. Hudson, and R. Tamassia. Accessing the internal organization of data structures in the JDSL library. In M. T. Goodrich and C. C. McGeoch, editors, *Algorithm Engineering and Experimentation (Proc. ALENEX '99)*, volume 1619 of *Lecture Notes Comput. Sci.*, pages 124–139. Springer-Verlag, 1999. 1, 10

[41] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java.* John Wiley & Sons, New York, NY, 1998. 1, 10, 11, 19

[42] T. Granlund. *The GNU Multiple Precision Arithmetic Library*, 1996. http://www.gnu.org/manual/gmp/ps/gmp.ps.gz. 9

[43] 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. 2, 20

[44] 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. 15

[45] A. Hausner and D. P. Dobkin. Making geometry visible: An introduction to the animation of geometric algorithms. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, chapter 8, pages 389–423. Elsevier Science Publishers, Amsterdam, The Netherlands, 2000. 38

[46] P. Heckbert, editor. *Graphics Gems IV.* Academic Press, Boston, MA, 1994. 5

[47] C. M. Hoffmann. The problems of accuracy and robustness in geometric computation. *IEEE Computer*, 22(3):31–41, 1989. 2, 20

[48] 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. 2, 20

[49] J. E. Hopcroft and P. J. Kahn. A paradigm for robust geometric algorithms. *Algorithmica*, 7(4):339–380, 1992. 2, 20

[50] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Object Technology Series. Addison-Wesley, Reading, MA, 1999. 22

[51] M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulations using rational arithmetic. *ACM Trans. Graph.*, 10(1):71–91, Jan. 1991. 20

[52] D. Kirk, editor. *Graphics Gems III.* Academic Press, Boston, MA, 1992. 5

[53] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing.* Addison Wesley, Reading, MA, 1993. 4

[54] M. Laszlo. *Computational Geometry and Computer Graphics in C++.* Prentice Hall, Englewood Cliffs, NJ, 1996. 6

[55] 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. 3, 21, 22

[56] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, MA, 2nd edition, 1999. 45

[57] G. Liotta, F. P. Preparata, and R. Tamassia. Robust proximity queries: An illustration of degree-driven algorithm design. *SIAM J. Computing*, 28(3):864–889, 1998. 2, 4, 20, 37, 38

[58] M. C. Loui et al. Strategic directions in research in theory of computing. *ACM Comput. Surv.*, 28(4):575–590, 1996. 1

[59] 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. *Comput. Geom. Theory Appl.*, 10(4):289–303, 1998. 1, 7, 8

[60] K. Mehlhorn and S. Näher. LEDA: A platform for combinatorial and geometric computing. *Commun. ACM*, 38(1):96–102, 1995. 1, 7

[61] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, England, 1999. 1, 7, 8

[62] K. Mehlhorn, S. Näher, M. Seel, R. Seidel, T. Schilz, S. Schirra, and C. Uhrig. Checking geometric programs or verification of geometric structures. *Comput. Geom. Theory Appl.*, 12(1–2):85–103, 1999. 5, 8

[63] S. Meyers. *More Effective C++*. Addison-Wesley, Reading, MA, 1996. 7

[64] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoret. Comput. Sci.*, 7(2):217–236, 1978. 15

[65] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, Reading, MA, 1996. 6, 11

[66] ObjectSpace. JGL, the Generic Collection Library for Java. http://www.objectspace.com/jgl/prodJGL.asp. 7

[67] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, Cambridge, England, 2nd edition, 1998. 6

[68] 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. 1, 8

[69] A. W. Paeth, editor. *Graphics Gems V*. Academic Press, Boston, MA, 1995. 5

[70] P. J. Plauger, A. A. Stepanov, M. Lee, A. Alexander, and D. R. Musser. *The Standard Template Libraries*. Prentice Hall, Upper Saddle River, NJ, 1998. 6, 11

[71] F. P. Preparata. A new approach to planar point location. *SIAM J. Comput.*, 10(3):473–482, 1981. 3, 21, 36

[72] F. P. Preparata. Planar point location revisited. *Internat. J. Found. Comput. Sci.*, 1(1):71–86, 1990. 3, 22

[73] 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. 5

[74] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, Reading, MA, 1998. 22

[75] S. Schirra. Robustness and precision issues in geometric computation. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, chapter 14, pages 597–632. Elsevier Science Publishers, Amsterdam, The Netherlands, 2000. 4

[76] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Comput. Geom.*, 18(3):305–363, 1997. 2, 20

[77] J. Snoeyink. Point location. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 30, pages 559–574. CRC Press, Boca Raton, FL, 1997. 3, 22

[78] K. Sugihara and M. Iri. A robust topology-oriented incremental algorithm for Voronoi diagrams. *Internat. J. Comput. Geom. Appl.*, 4(2):179–228, 1994. 2, 20

[79] Sun. Java 2 SDK, Standard Edition. http://java.sun.com/products/jdk/1.2/. 7

[80] A. Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479, 1996. 10, 43

[81] R. Tamassia. On-line planar graph embedding. *J. Algorithms*, 21(2):201–239, 1996. 15

[82] R. Tamassia et al. Strategic directions in computational geometry. *ACM Comput. Surv.*, 28(4):591–606, 1996. 1, 4

[83] P. van Oosterom. *Reactive Data Structures for Geographic Information Systems.* Oxford University Press, Oxford, England, 1993. 4, 16

[84] P. Wegner. Concepts and paradigms of object-oriented programming. *ACM OOPS Messenger*, 1(1):7–87, 1990. 10, 43

[85] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 35, pages 653–668. CRC Press, Boca Raton, FL, 1997. 2, 20

[86] C. K. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7(1):3–23, 1997. 2, 4, 20

[87] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific Press, Singapore, 2nd edition, 1995. 2, 4, 20

A Object-Oriented Concepts

The design of GeomLib relies on various object-oriented design concepts (see, e.g., Refs. [16, 35, 80, 84]). In this section, we 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 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 Ref. [16].

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 and/or in the number of parameters.

If methods with the same name, and type 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 5 on page 6).

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 objects participating in the solution, their roles and the way they interact with each other. Collections of systematically described design patterns (see e.g., Ref. [35]) 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.

Java compilers produce an intermediate, language independent byte code that is comparable to the intermediate code produced by most compilers. (In fact, various compilers have been introduced for other languages to produce Java byte code.) A number of aspects distinguish this byte code from similar formats: it is portable, to the extent that it can be loaded 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, 56].

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. Recently, however, Sun has released 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.

C Code Fragments

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

  // class variable(s)

  protected static GeomTester2D gt_ = new GeomTester2DImpl();

  // instance variable(s)

  protected String label_;

  // constructor(s)

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

  // instance method(s)

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

  // instance method(s) from java.lang.Object

  public String toString () {
    return label_;
  }

}
```

```
public class FaceRegion extends AbstractRegion {

  // constructor(s)

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

}
```

```
public class HorEdgeRegion extends AbstractRegion {

  // constructor(s)

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

}
```

Code Fragment 7: Abstract class `AbstractRegion`, and classes `FaceRegion` and `HorEdgeRegion`.

```
public class EdgeRegion extends AbstractRegion implements Separator {

  // instance variable(s)

  protected LinearCurve2D linearCurve_;

  // constructor(s)

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

  // instance method(s)

  public DiscrimResult whichSide (Point2D q) {
    return new DiscrimResultImpl(gt_.leftRight(q,linearCurve_),this);
  }

}
```

```
public class VertexRegion extends AbstractRegion implements Separator {

  // instance variable(s)

  protected Point2D point_;

  // constructor(s)

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

  // instance method(s)

  public DiscrimResult whichSide (Point2D q) {
    return new DiscrimResultImpl(gt_.leftRight(q,point_),this);
  }

}
```

Code Fragment 8: Classes `EdgeRegion` and `VertexRegion`.

```
public class HorLine extends AbstractRegion implements Separator {

  // instance variable(s)

  protected Point2D point_;
  protected Region onRegion_;

  // constructor(s)

  public HorLine (Point2D point, String label, Region onRegion) {
    super(label);
    point_ = point;
    onRegion_ = onRegion;
  }

  // instance method(s)

  public DiscrimResult whichSide (Point2D q) {
    return new DiscrimResultImpl(gt_.aboveBelow(q,point_),onRegion_);
  }

}
```

```
public class DiscrimResultImpl implements DiscrimResult {

  // instance variable(s)

  protected int side_;
  protected Region region_;

  // constructor(s)

  public DiscrimResultImpl (int side, Region region) {
    side_ = side;
    region_ = region;
  }

  // instance method(s)

  public int side () {
    return side_;
  }

  public Region region () {
    return region_;
  }

}
```

Code Fragment 9: Classes `HorLine` and `DiscrimResultImpl`.

```
public class HorInterval extends AbstractRegion implements Separator {

  // instance variable(s)

  protected Point2D leftPoint_;
  protected Point2D rightPoint_;
  protected Region leftPointOnRegion_;
  protected Region rightPointOnRegion_;
  protected Region onRegion_;

  // constructor(s)

  public HorInterval (Point2D leftPoint, Point2D rightPoint, String label, Region leftPointOnRegion,
  Region rightPointOnRegion, Region onRegion) {
    super(label);
    leftPoint_ = leftPoint;
    rightPoint_ = rightPoint;
    leftPointOnRegion_ = leftPointOnRegion;
    rightPointOnRegion_ = rightPointOnRegion;
    onRegion_ = onRegion;
  }

  // instance method(s)

  public DiscrimResult whichSide (Point2D q) {
    int leftTest;
    int rightTest;
    if (leftPoint_ == Point2D.INFINITE)
      leftTest = GeomTester2D.POSITIVE;
    else
      leftTest = gt_.leftRight(q,leftPoint_);
    switch (leftTest) {
    case GeomTester2D.NEGATIVE:
      return new DiscrimResultImpl(GeomTester2D.NEGATIVE,this);
    case GeomTester2D.ON:
      return new DiscrimResultImpl(GeomTester2D.ON,leftPointOnRegion_);
    case GeomTester2D.POSITIVE:
    default:
      if (rightPoint_ == Point2D.INFINITE)
        rightTest = GeomTester2D.NEGATIVE;
      else
        rightTest = gt_.leftRight(q,rightPoint_);
      switch (rightTest) {
      case GeomTester2D.NEGATIVE:
        return new DiscrimResultImpl(GeomTester2D.ON,onRegion_);
      case GeomTester2D.ON:
        return new DiscrimResultImpl(GeomTester2D.ON,rightPointOnRegion_);
      case GeomTester2D.POSITIVE:
      default:
        return new DiscrimResultImpl(GeomTester2D.POSITIVE,this);
      }
    }
  }

}
```

Code Fragment 10: Class `HorInterval`.