Chapter 6

Designing Geometric Algorithms

This chapter is joint work with Vikas Kapoor and Dietmar Kühl.

The design phase of an implementation is of utmost importance for its efficiency, flexibility, and ease-of-use. While focusing on flexibility, we demonstrate that there is no reason for sacrificing ease-of-use. Furthermore, we show that a gain in flexibility does not necessarily cause a loss in efficiency.

In this chapter we present a design concept for geometric algorithms that is based on the *generic programming* paradigm. Generic programming is about making implementations more flexible by making them more general. Abstracting from concrete in- or output data representation is an example of generic programming. In contrast to normal programs, the parameters of generic programs are often quite rich in structure. Such parameters might be other programs, types or type constructors, or even other programming paradigms [BS98].

The generic programming paradigm has been so successful that a model—the *Standard Template Library* (STL) [MS96]—was created and added to C++, currently one of the most popular programming languages. The STL is a library of generic components, i.e. of algorithms, data containers, and *iterators* mediating between the former two. Iterators help to decouple algorithms from the type of data container they operate on. While iterators have been known before, the real novelty of the STL was the introduction of a requirements-based taxonomy of iterators, which gives a guideline for full decoupling, and an implementation of this taxonomy using the C++ template mechanism. By becoming part of the C++ standard, the STL has attracted considerable attention and has itself set a standard for good design.

After the introduction of the STL further concepts such as *data accessors* have been suggested in the C++ literature to help programmers make their implementations even more generic [Küh96, Wei97]. Data accessors are a means to further decouple an implementation from the representation of in- and output data [KW97]. So far, these extensions have been applied predominantly to graph problems [NW96]. Exceptions such as [Wei98, Ket98] deal with the representation of geometric objects, not with the design of geometric *algorithms*, our main interest here. In order to show the relevance of STL-style generic programming including later extensions as data accessors for geometric algorithms, we investigate a simple rectangle-intersection algorithm that follows the well-known sweep-line paradigm.

Using this example we lead the reader step by step from an inflexible, naive interface to a truly flexible interface that supports code reuse. These steps reflect our own change of perspective during the implementation of our label-placement algorithms. Note that reusability helps to lower implementation costs in the long run and to achieve correctness—for the simple reason that more users mean more testing. The reader should be familiar with the programming language C++ that we chose to demonstrate our concept.

In order to support our concept with experimental data, we implemented a sweep-line algorithm for the rectangle intersection problem in C++ in two ways; (a) using straight-forward object orientation and (b) following our design concepts. We compare the runtime of both implementations, as well as the size of their source code and executables. The data we used for the runtime comparison stems from random generators as well as from real world instances. The example classes have been used for an experimental analysis of map-labeling algorithms and are described in detail in [WW98]. The source code of our implementations, the documentation of the interfaces, the test data and its description are accessible via the WWW[1].

While the ingredients of our concept have already been known, to our knowledge this is the first time that they are applied so rigorously to a geometric problem, that they are made accessible in the form of a tutorial and that they are accompanied by a thorough experimental analysis.

Other than the STL, the C++ Library of Efficient Data Types and Algorithms (LEDA) [NM90] was designed having mostly ease-of-use rather than genericity in mind. The resulting short-comings in terms of interfacing with user-defined data structures are investigated in [Küh96]. This has led to the implementation of a LEDA extension package for graph iterators [NW96].

A recent report about the design of CGAL, the Computational Geometry Algorithms Library [Ove96], also a C++ library, includes a section about the pros and cons of generic versus object-oriented programming [FGK$^+$98].

This chapter is structured as follows. In Section 6.1, we describe our example algorithm for the rectangle-intersection problem. In Section 6.2, we present a naive interface for this algorithm, investigate its disadvantages and modify it step-by-step to a generic and thus flexible interface. Finally, in Section 6.3, we compare the implementations of the naive and the most flexible interface of the previous section.

[1]see `http://www.math-inf.uni-greifswald.de/map-labeling/design/`

6.1 Algorithm

We demonstrate our design concepts at the example of a sweep-line algorithm for detecting all intersections among a set of axis-parallel rectangles in the plane [Ede80]. Our sweep line will be a vertical line sweeping the plane from left to right. As usual, the sweep line is supported by two data structures, the *event-point schedule* and the *sweep-line status*.

Event points are the x-values where the sweep line must stop because either its status changes or intersections have to be reported. In our case all event points are known before the sweep begins: they are the x-values of the left and right edges of the rectangles. Thus a sorted list suffices to implement our event-point schedule. An event point must be stored in such a way that it is clear whether it refers to a left or right edge of a rectangle.

The sweep-line status stores intervals corresponding to the intersections of the sweep line with the given rectangles. The endpoints of the intervals are the y-values of the lower and upper edges of the input rectangles. Initially the sweep-line status is empty. When a left edge of a rectangle is encountered during the sweep, the interval corresponding to the edge is inserted into the sweep-line status. A rectangle is reported if its interval is currently in the sweep-line status and intersects the new interval. When a right edge of a rectangle is encountered, the corresponding interval is deleted from the sweep-line status.

This reduces the rectangle intersection problem to the problem of maintaining a set of intervals such that intervals can be efficiently inserted and deleted, and interval-intersection queries can be answered quickly. To achieve this, we implement our sweep-line status with the interval-tree data structure [Ede80]. We have implemented a semi-dynamic version, which must be initialized with the endpoints of all intervals it is going to contain during the sweep. The preprocessing, namely sorting the points and building up an empty balanced tree, takes $O(n \log n)$ time, where n is the number of intervals. Inserting intervals can then be done in $O(\log n)$ time, deleting in constant time, while a query takes $O(k + \log n)$ steps, where k is the number of intervals reported. Since every interval is stored only once in the interval tree, the storage consumption is linear.

6.2 Step by Step Towards Good Design

In this section we start with a naive interface for the algorithm described in the previous section. Then we consider the limitations of this approach and show how these can be overcome. We will improve the straight-forward solution in several steps, each of which is independent of the others. Note that the naive interface could easily be implemented without explicit data conversion by all of the subsequent solutions—if that was our goal.

6.2.1 The Naive Approach

How would a naive programmer interface an algorithm for the rectangle intersection problem described above? He might implement a function, whose input would be an array of rectangles and their number. The output could either consist of a list of pairs of intersecting rectangles, or, more user-friendly, of a so-called conflict graph. In this graph, each rectangle is represented by a node, and two nodes are connected by an edge, if the corresponding rectangles intersect. This is the interface described in Program 6.2.1.

```
class Rectangle;
class Graph;
Graph* rectangle_intersection(Rectangle* rectangle_array, int n);
```

Program 6.2.1: a naive functional interface

This interface would force the programmer to implement the data structures `Rectangle` and `Graph`. On the other hand, it would force the user in most cases to convert his rectangles into those required by the interface, and, after calling `rectangle_intersection`, extract the desired information from the conflict graph. In order to do so, the user would have to study the interfaces of the in- and output data structures. Another disadvantage of the naive interface is that it would not even allow the user to hand over rectangles in a *container* different from an array, like a list or a set.

6.2.2 Decoupling Algorithm and Data Organization

The most obvious disadvantage of our previous interface is the tight link between the algorithm and its in- and output data structures. In order to decouple algorithm and data organization, we must solve two problems.

(P1) *container independence*, i.e. we do not want to force the user to hand over an *array* of rectangles.

(P2) *representation independence*, i.e. we should not require the use of a fixed representation of rectangles or graphs, but rather accept any representation fulfilling certain requirements.

The first problem can be solved with the help of *iterators.* Iterators are a generalization of pointers; they are light-weight objects that point to other objects. As the name suggests, iterators are used to iterate over a range of objects: if an iterator points to an element in a range, it can be incremented so that it points to the next element or to an end-of-range marker. Iterators can also be tested for equality, e.g. to test whether the end of a range is reached. They represent an extremely versatile link between containers and algorithms. If an algorithm's interface takes iterators as arguments, then the algorithm can be applied to any container that provides access to its elements via iterators. This is the central concept introduced by the STL [MS96].

Consequently, our next interface will expect iterators to manage the input. Handling the output via iterators is not so simple, as the conflict graph is not a linear structure. This problem is attacked in the following section. For the time being, we ask the user to provide his definition of a graph as a template parameter to our interface. Of course, this definition must fulfill some requirements. The implementation expects the member functions for inserting nodes and edges.

```
class User_Rectangle;
class User_graph<User_Rectangle*>;
typedef User_graph<User_Rectangle*> Graph;
// requirements
typedef Graph::node_type node;
node Graph::insert_node (User_Rectangle*);
void Graph::insert_edge (node&, node&);
```

Note that the user is not forced to write his own graph data structure; he can use that of any library and write a simple wrapper that implements our requirements with the help of the library graph.

In order to solve the second problem, we use so-called *data accessors*. While iterators realize access to objects, data accessors are used to access the data associated with these objects [KW97]. Data accessors have two parts, a data accessing function, which is responsible for the actual access, and a light-weight object. This object is also referred to as the data accessor. It encapsulates the data type to be accessed and is used to select the correct data accessing function. Our next interface will require the user to provide such data accessors for his representation of the input data. Assume that the user declares the following `User_Rectangle` type.

```
typedef CoordType double;
struct  User_Rectangle { CoordType llx, lly, urx, ury; };
typedef User_Rectangle* User_iterator;
```

where `llx`, `lly`, `urx`, and `ury` represent the coordinates of the lower left and upper right corner of the rectangle, respectively. The data accessor for the x-coordinate of the lower left corner can then be defined as follows.

```
// data accessor
struct LLXDA { typedef CoordType value_type };
// data accessing function
CoordType get(LLXDA const& da, User_iterator const& it)
{ return (*it).llx; }
```

All our algorithm needs to know about the user's rectangles are the coordinates of their corners. This is exactly what the data accessors will supply. Note the interplay between data accessing function and its arguments, namely the data accessor and the iterator, in Program 6.2.2.

```
template < class Iterator, class Graph,
           class LLXDA, class LLYDA, class URXDA, class URYDA >
void rectangle_intersection(Iterator begin, Iterator end,
                            Graph& graph,
                            LLXDA llxda, LLYDA llyda,
                            URXDA urxda, URYDA uryda)
{
   for (Iterator rect_it = begin; rect_it != end; ++rect_it)
     typename LLXDA::value_type lower_x = get(llxda, rect_it);
   // ...
   Iterator rect_it1, rect_it2;      // ...
   typename Graph::node_type node1 = graph.insert_node(rect_it1),
                             node2 = graph.insert_node(rect_it2);
   graph.insert_edge(node1, node2);  // ...
}
```

Program 6.2.2: a functional, data-organization independent interface

6.2.3 Tightening Control

Suppose the user of our implementation is only interested in a tiny fraction of the output, like the number of intersections or all rectangles intersecting a given rectangle. Or suppose he would like to abort the execution of the algorithm when the sweep line reaches a certain x-value or if another condition becomes true. With the interface suggested in the previous subsections, he would not be able to take advantage of such a situation. It is clear that a functional implementation cannot provide such a degree of interaction between the user and the algorithm. Thus we will switch to a class interface, which allows us to have a state and offer the user more information about the algorithm's progress.

The key to more control is the *loop kernel* [Küh96, Wei97]. The loop kernel is a method which encapsulates the body of the central loop of the algorithm. It can be advanced in single steps and informs the user about the current state of the algorithm. The loop kernel makes the whole algorithm look like an iterator that can be incremented until the execution is finished.

For our example algorithm, we would define the following states: `none` at the beginning, `done` at the end, `rectangle_begin` when the sweep line hits the left edge of a rectangle, and `rectangle_end` when a right edge is reached. We implement the loop kernel by a member function `step()` that advances the sweep line to the next event point and returns the current state, see Program 6.2.3.

The concept of the loop kernel would be incomplete without the idea of *full logical inspectability.* An algorithm is fully logically inspectable if the user can access all important intermediate results during the execution. This is important for animation, interaction or simply for debugging. In our example this would mean access to the content of the whole sweep-line status, not just to those rectangles that intersect the current one. Hence we offer two pairs of iterators, one referring to the whole sweep-line status, and one marking just the range of current intersections. Their type is discussed in Section 6.2.5.

```
template < class Iterator,
           class LLXDA, class LLYDA, class URXDA, class URYDA >
class Rectangle_intersection {
public:
   // constructor
   Rectangle_intersection(Iterator begin, Iterator end);
   // return the result in user supplied graph
   template <class Graph> void run(Graph& graph);
   // loop kernel
   enum  state   { none, rectangle_begin, rectangle_end, done };
   bool  valid() { return (state != done); }
   state step();
   // full logical inspectability
   Iterator current(); // rectangle represented by curr. event point
   typedef  /* ... */  Solution_iterator;
   Solution_iterator begin();
   Solution_iterator end();
   // queries: report all rectangles intersecting the curr. rectangle
   Solution_iterator current_begin();
   Solution_iterator current_end();
};
```

Program 6.2.3: a class interface

Note that the user can still get the output in a graph representation as before, but the existence of a graph data structure is no longer a prerequisite to using the algorithm. (This can also be achieved without templated member functions like `run()`, which are standard conform, but not yet supported by all C++ compilers.)

6.2.4 Influencing Critical Decisions

Another important question is the following. Which definition of "intersection" do we implement? Does touching already imply an intersection? What about inclusion? Of course, the answers to these questions will differ from application to application. Instead of trying to cover all possible interpretations, we leave the definition of an intersection to the user. To do so, we must isolate the decision making parts of our implementation such that no information local to the algorithm is needed. Then the user can provide *function objects*, with which the algorithm is parameterized.

Our rectangle intersection algorithm has two basic data structures, the event-point schedule and the sweep-line status. The order of event points decides, whether the projections of the corresponding rectangles intersect on the x-axis. Similarly, a query of the sweep-line status returns rectangles, whose projections intersect that of the current rectangle on the y-axis. To differentiate between these two categories of intersections, we require the user to provide two function objects, one that enables us to sort the event-point schedule accordingly and the other for determining the behavior of the interval tree that implements the sweep-line status.

In the following we give examples of function objects that view rectangles as topologically open and thus do not report rectangles touching each other. To sort the event points accordingly, we just have to make sure that an event point e_l corresponding to the left edge of a rectangle will be inserted into the schedule after all event points that belong to right edges with the same x-coordinate. Then all of the latter rectangles are already removed from the sweep-line status and will not be reported when we reach e_l.

Since an event point is internal to our algorithm, it cannot be accessed directly by the user. Thus we have to isolate the information needed to sort the event points. If two event points have the same x-value, we need to know whether each corresponds to a left or right edge of the respective rectangle. This information can easily be obtained via the x-coordinate of the event point plus the iterator pointing to the corresponding rectangle. Their types are of course known to the user. Thus we require a function object, which realizes a comparison between two pairs of the corresponding types, see Program 6.2.4.

```
class Compare_x {
public:
   typedef pair<CoordType, User_iterator> Pair;
   bool operator()(const Pair& p, const Pair& q) {
     bool before = true;
     if (p.first == q.first)
     {
       // p is the x-coordinate of the left edge of a rectangle
       if (p.first == p.second->llx) before = false;
     }
     else before = (p.first < q.first);
     return before;
   }
};
```

Program 6.2.4: compare function object for sorting the event-point schedule

The function object for the interval tree is quite simple, see Program 6.2.5.

```
class Compare_y {
public:
    bool operator()(CoordType const y1, CoordType const y2)
    { return (y1 < y2); }
};
```

Program 6.2.5: compare function object for querying the sweep-line status

Note the difference of this technique to the approach of implementing the most general definition of intersection and then filtering out all undesired information. Our function objects have the potential to reduce the complexity not just of the *output*, but also of the *computation*.

6.2.5 The Complete Interface

Program 6.2.6 shows the interface resulting from our successive improvements. At first sight it might look more complicated than the naive interface. To guarantee a smooth learning curve for users not familiar with generic programming and the algorithm we implemented, a good library would provide a concrete representation of rectangles and graphs, say `Our_rectangle` and `Our_graph`, as well as defaults for the other template parameters. This would make it possible to use our algorithm as in Program 6.2.7.

Note that the constructor has been parameterized by objects of each of the class's template parameters. This allows the user to instantiate our algorithm with objects that may be constructed other than by their default constructor. The data accessors `URXDA` and `URYDA` for the coordinates of the upper right corner of the input rectangles might for example be parameterized with a given height and width of the rectangles, which could change from instantiation to instantiation of the intersection algorithm.

Of course, we have also implemented the interval tree for the sweep-line status with the concepts presented here. This is the point where the flexibility of our algorithms bears fruit, since we do not have to convert any rectangles into intervals to construct the interval tree. This is due to the fact that the endpoints of the intervals we want to store in the tree correspond to the y-coordinates of the corners of our input rectangles. Thus we just parameterize the nested interval tree class with a subset of the template parameters of the class `Rectangle_intersection`. The required parameters are the types of the rectangle iterator and the compare function object for y-coordinates as well as the data accessors `LLYDA` and `URYDA`, see the private definition of the type `Interval_tree` in Program 6.2.6. So in a way, the class `Interval_tree` considers our input rectangles to be nothing but intervals, namely the rectangles' projection on the y-axis.

The iterator `Solution_iterator` needed to traverse the sweep-line status is provided by the class `Interval_tree`. Dereferencing a `Solution_iterator` supplies the user with an iterator of the type, with which he has parameterized the class `Rectangle_intersection`.

Program 6.2.8 shows how the data structures required by our algorithm could be declared. Our example demonstrates one of the advantages of using data accessors. If the input consists of squares of common size, the user has to store only the coordinates of their lower left corners. When our algorithm needs a coordinate of the opposite corner, the corresponding data accessing function computes it on the fly. This reduces the storage consumption of the input by 50%.

Note that not all compilers support the pointer-to-member mechanism for template parameters we used here. The obvious workaround is to declare explicitly all four data accessors and accessing functions required by the generic implementation.

The intersection algorithm for squares can then be declared as in Program 6.2.9.

```
class Our_rectangle; class Our_graph;
class Our_lxda; class Our_lyda; class Our_hxda; class Our_hyda;
class Our_compare_x; class Our_compare_y;

template < class Iterator = Our_rectangle*,
           class LLXDA = Our_lxda, class LLYDA = Our_lyda,
           class URXDA = Our_hxda, class URYDA = Our_hyda,
           class CompareX = Our_compare_x,
           class CompareY = Our_compare_y >
class Rectangle_intersection
{
   // type of the sweep-line status
   typedef Interval_tree<Iterator, LLYDA, URYDA, CompareY>
     Sweep_line_status;
public:
   // type of rectangle coordinates
   typedef typename LLXDA::value_type value_type;
   // type of iterator used to return intersections
   typedef typename Sweep_line_status::iterator Solution_iterator;
   // constructor
   Rectangle_intersection(Iterator begin, Iterator end,
                          LLXDA lxda = LLXDA(), LLYDA lyda = LLYDA(),
                          URXDA hxda = URXDA(), URYDA hyda = URYDA(),
                          CompareX comp_x = CompareX(),
                          CompareY comp_y = CompareY() );
   // loop kernel
   enum  state { none, rectangle_begin, rectangle_end, done };
   bool  valid() const { return (state != done); };
   state step();
   // full logical inspectability
   Iterator current(); // rectangle represented by current event point
   value_type current_sweep_line_position() const;
   Solution_iterator begin();
   Solution_iterator end();
   // queries: report all rectangles intersecting the curr. rectangle
   Solution_iterator current_begin();
   Solution_iterator current_end();
   // miscellaneous member functions
   int number_of_intersections() const;
   // return conflict graph of input rectangles
   template <class Graph> void run (Graph& graph);
};
```

Program 6.2.6: flexible interface of the class `Rectangle_intersection`

```
   Our_rectangle rects[10];
   Our_graph our_graph;
   // ...
   // declare and run the rectangle intersection algorithm
   Rectangle_intersection rectangle_intersection(rects, rects+10);
   rectangle_intersection.run(our_graph);
```

Program 6.2.7: Ease-of-use: applying `Rectangle_intersection` to library-supplied data structures

```
// representation of a square
typedef int CoordType;
const  CoordType length = 50;
struct Square { CoordType x, y; };

// data accessors for the above representation
template<CoordType Square::*member>
struct CoordLowDA  { typedef CoordType value_type; };
template<CoordType Square::*member>
struct CoordHighDA { typedef CoordType value_type; };

// data accessing functions
template <CoordType Square::*member>
inline CoordType get(CoordLowDA<member>  const&,
                     User_iterator const& it)
{ return (*it).*member; };

template <CoordType Square::*member>
inline CoordType get(CoordHighDA<member> const&,
                     User_iterator const& it)
{ return (*it).*member + length; };

// compare function objects as defined above
class Compare_x;  class Compare_y;
```

Program 6.2.8: Flexibility: user-supplied types for using the class `Rectangle_intersection`

```
// algorithm
typedef Rectangle_intersection< User_iterator,
                                CoordLowDA<&(Square::x)>,
                                CoordLowDA<&(Square::y)>,
                                CoordHighDA<&(Square::x)>,
                                CoordHighDA<&(Square::y)>,
                                Compare_x, Compare_y >
        Square_intersection_algo;

// iterator for access to the solution
typedef typename Square_intersection_algo::Solution_iterator
                 Solution_it;
```

Program 6.2.9: declaration of the class `Rectangle_intersection` for squares of common size

Program 6.2.10 shows how the types declared in Program 6.2.8 and 6.2.9 are plugged into our interface.

```
main()
{
  Square squares[10];  // input of squares...
  Square_intersection_algo my_algo(squares, squares+10);
  while (my_algo.valid())
    if (my_algo.step() == Square_intersection_algo::rectangle_begin)
    {
      Square* curr = my_algo.current();
      // assuming output operator for Square
      cout << *curr << " intersects: " << endl;
      Solution_it end = my_algo.current_end();
      for (Solution_it it = my_algo.current_begin(); it != end; ++it)
        cout << *it << endl;
    }
}
```

Program 6.2.10: a toy application for user-supplied data structures

6.3 Experiments

We compare two different implementations of the rectangle intersection problem in terms of runtime. The implementations are characterized as follows.

1. the *object-oriented* approach encapsulates the interface of Section 6.2.1 in a class. Its implementation requires a fixed type of rectangle. Similarly, the underlying interval tree is a class requiring a fixed type of interval.
2. the *generic* approach implements the generic interface of Section 6.2.5. All data structures are implemented according to the concepts suggested in this chapter.

6.3.1 Example Classes

We ran both implementations on the following eight example classes. These benchmarks have also been used to compare the quality of map-labeling algorithms experimentally, see Section 3.2. They are available from our Web page.

RandomRect. We choose n points uniformly distributed in a square of size $25n \times 25n$. Each point corresponds to the lower left corner of a rectangle. To determine the size of each rectangle, we choose the length of both edges independently under normal distribution, take its absolute value and add 1 to avoid non-positive values. Finally we multiply both rectangle dimensions by 10.

DenseRect. Here we try to place as many rectangles as possible on an area of size $\alpha_1\sqrt{n} \times \alpha_1\sqrt{n}$. α_1 is a factor chosen such that the number of successfully placed rectangles is approximately n, the number of sites asked for. We do this by randomly selecting the rectangle size as above and then trying to place the rectangle 50 times. If we don't manage, we select a new rectangle size and repeat the procedure. If none of 20 different sized rectangles could be placed, we assume that the area is well covered, and stop. For each rectangle we placed successfully, we return its height and width. To generate (a limited amount of) intersections, we randomly choose a corner and use that as the position of the lower left corner of the rectangle we return.

RandomMap and DenseMap. These example classes try to imitate a real map using the same methods as RandomRect and DenseRect for placing the lower left corner of the rectangles, but more realistic rectangle sizes. We assume a distribution of 1:5:25 of cities, towns and villages. After randomly choosing one of these three classes according to the assumed distribution, we set the rectangle height to 12, 10 or 8 points accordingly. The length of the rectangle text then follows the distribution of a set of 377 German Railway station names. We assume a typewriter font and set the rectangle length to the number of characters times the font size times 2/3. The multiplicative factor reflects the ratio of character width to height.

VariableDensity. This example class was suggested in an experimental comparison of map-labeling algorithms by Christensen et al. [CMS95]. There, the points are distributed uniformly on a rectangle of size 792×612. All rectangles are of equal size, namely 30×7.

HardGrid. In principle we use the same method as for DenseRect and DenseMap, that is, trying to place as many rectangles as possible into a given area. In order to do so, we use a grid of $\lfloor \alpha_2 \sqrt{n} \rfloor \times \lceil \alpha_2 \sqrt{n} \rceil$ cells with edge lengths n. Again, α_2 is a factor chosen such that the number of successfully placed squares is approximately n. In a random order, we try to place a square of edge length n into each of the cells. This is done by randomly choosing a point within the cell and putting the lower left corner of the square on it. If it overlaps any of the squares placed before, we repeat at most 10 times before we turn to the next cell. Finally, we choose a random corner of the square we placed and use that as the lower left corner of the square we return.

RegularGrid. We use a grid of $\lfloor \sqrt{n} \rfloor \times \lceil \sqrt{n} \rceil$ square grid cells. For each cell, we randomly choose a corner and place a point with a small constant offset near the chosen corner. On this point, we place a square with an edge length of grid cell size minus the offset.

MunichDrillholes. The municipal authorities of Munich provided us with the coordinates of roughly 19,400 ground-water drill holes within a 10 by 10 kilometer square centered approximately on the city center. From these sites, we randomly pick a center point and then extract a given number of sites closest to the center point according to the maximum norm. Thus we get a rectangular section of the map. Its size depends on the number of points asked for. The drill-hole labels are abbreviations of fixed length. By scaling the x-coordinates, we make these rectangular labels into squares and subsequently apply an exact solver for label size maximization. The label size determined in this way is the size of the squares we return. We place them with their lower left corner on the scaled drill holes.

Figures 6.17 to 6.24 show an important parameter of these example classes, namely their average number of intersections. We did not count pairs of *touching* rectangles, like in the example implementation of the compare function objects in Section 6.2.4. We used examples of approximately 250, 500,... up to 3000 rectangles. For each of the example classes and each example size, we averaged the runtime and the number of intersections over 30 files.

6.3.2 Results

In Figures 6.1 to 6.24, the average number of rectangles over the 30 files for each example size is shown on the x-axis. On the y-axis, Figures 6.1 to 6.8 show the average runtime for both implementations for the interesting special case that the user is merely interested in the number of intersections. This is slightly favorable for the generic implementation since there, no information about which rectangles are in fact intersecting, has to be stored. In this setting, the running times were nearly identical.

Figures 6.9 to 6.16 show the runtime when all intersections are stored in adjacency lists during the execution of the two programs. Here the generic implementation took between 0 and 60% longer than the object-oriented version. Note however that this gap is smaller—not more than 20%—when the two example classes RegularGrid and MunichDrillholes with the lowest density are ignored, i.e. those with the smallest ratio of the number of rectangles and the number of intersections, see Figure 6.22 and 6.23.

The average runtime is given in CPU seconds. It was measured on a Sparc-Ultra-1 machine; the programs were compiled with the SUN CC-4.2 compiler with optimizer option `-fast`. On our Web page, we provide graphs for the same test suite run on an SGI IP27 with the mipsPRO CC-7.1 compiler. The results were comparable.

While the generic and the object-oriented implementation do differ much in their source code size, we listed the sizes of executables for identical test programs for both implementations in Table 6.1. The source code is available via our Web page as well.

test program for ...	*size of executable in KBytes*
object-oriented interval tree	63
generic interval tree	74
object-oriented rectangle intersection	128
generic rectangle intersection	120

Table 6.1: sizes of the executables

It is interesting to note that although the executable of a simple test program for the generic version of the interval tree is slightly larger than that of its object-oriented counterpart, it is opposite for the corresponding versions of the rectangle intersection data structure. The reason for this seems to be that the generic implementation does not need to convert and store the input data for the interval tree explicitly. This difference in the sizes of the executables may become substantial in case of larger class hierarchies.

6.3.3 Evaluation

We have presented a toolbox of concepts which helps to turn inflexible into generic and thus reusable interfaces. We have exemplified this transition at a geometric algorithm, namely a sweep-line algorithm for the rectangle intersection problem. On the road from a naive to a flexible interface for this algorithm, we suggested to decouple algorithms from the organization of their in- and output data. Then we presented the loop kernel as an important means of gaining control over the execution of an algorithm. Full logical inspectability introduced additional transparence. Finally, we came up with function objects as a way to parameterize algorithms with information that can be used to influence critical decisions.

In our experiments, we compared a generic to an object-oriented implementation of the rectangle intersection algorithm. We investigated the runtime of the two implementations on eight example classes from random and real world sources and in two different settings. In the first setting, which was favorable for the generic implementation, the running times were nearly identical. In the second setting, the generic implementation was just 20% slower than its competitor on all example classes but the two least dense. We do not think that this is too high a price for the gain of flexibility achieved by the generic interface.

The slowdown caused by an object-oriented library like LEDA is of a different order of magnitude. In [MN92], the runtime of Dijkstra's algorithm using LEDA Fibonacci heaps is compared to an implementation using special integer Fibonacci heaps. For this example, the authors report a slowdown by a factor of 3.

Runtimes for Computing the Number of Intersections

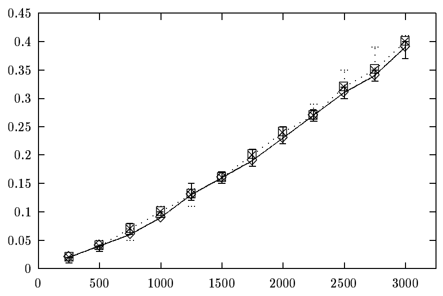

Figure 6.1: RandomMap

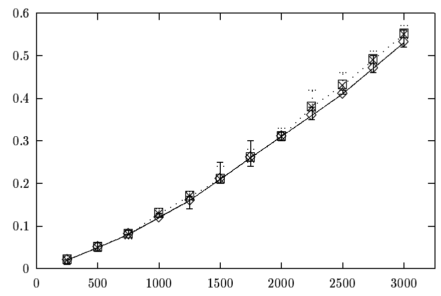

Figure 6.2: RandomRect

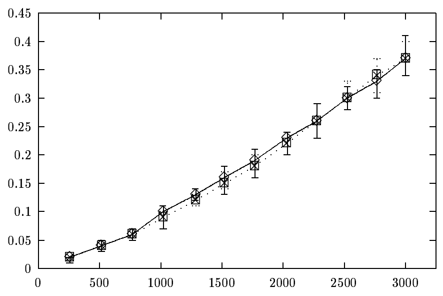

Figure 6.3: DenseMap

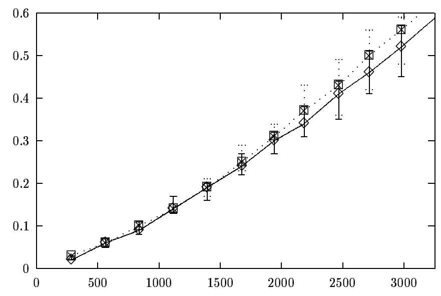

Figure 6.4: DenseRect

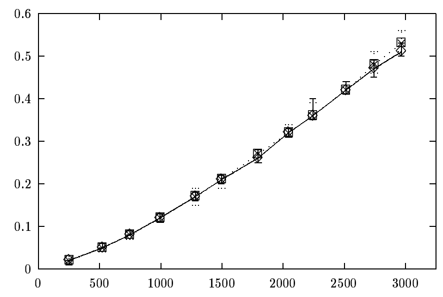

Figure 6.5: HardGrid

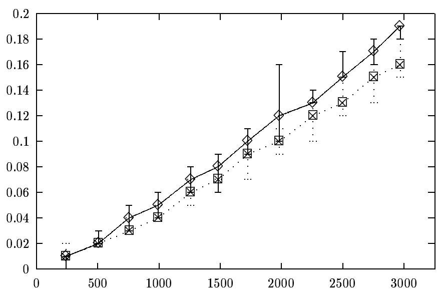

Figure 6.6: RegularGrid

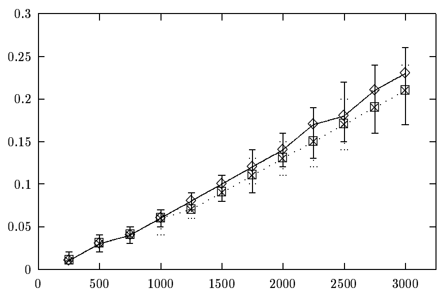

Figure 6.7: MunichDrillholes

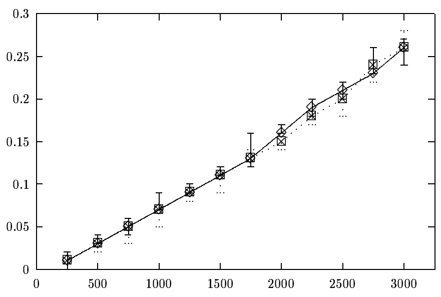

Figure 6.8: VariableDensity

Generic Implementation ◇ Object-Oriented Implementation ⊠

Runtimes for Generating Adjacency Lists

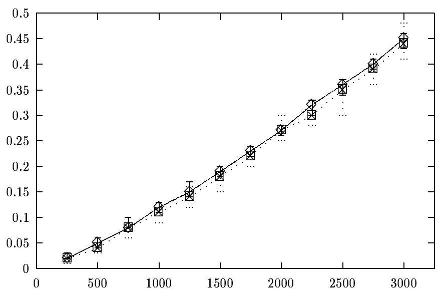

Figure 6.9: RandomMap

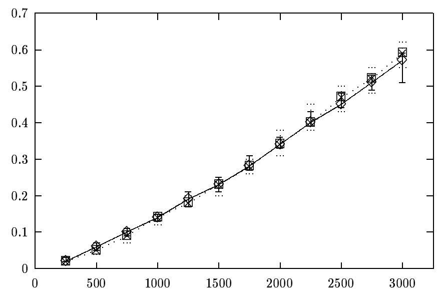

Figure 6.10: RandomRect

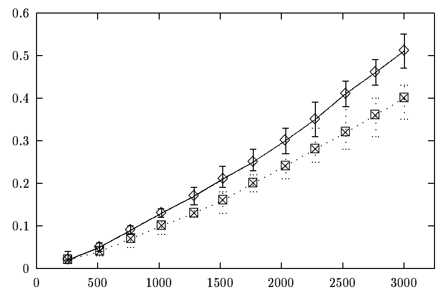

Figure 6.11: DenseMap

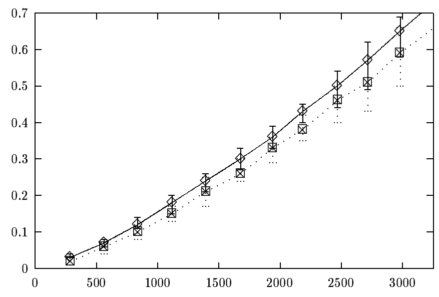

Figure 6.12: DenseRect

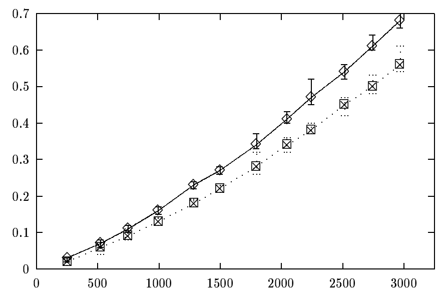

Figure 6.13: HardGrid

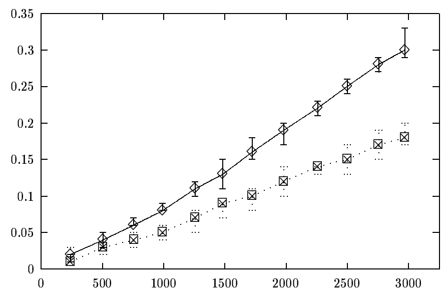

Figure 6.14: RegularGrid

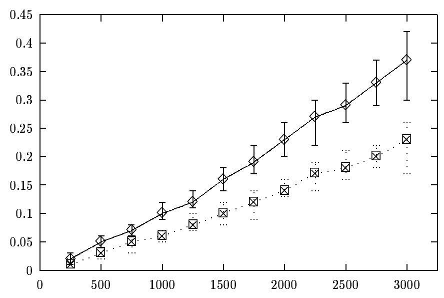

Figure 6.15: MunichDrillholes

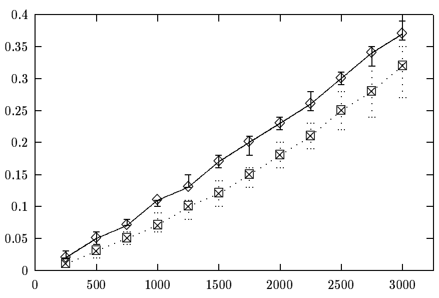

Figure 6.16: VariableDensity

Generic Implementation —◇— Object-Oriented Implementation ···⊠···

Numbers of Intersecting Rectangles

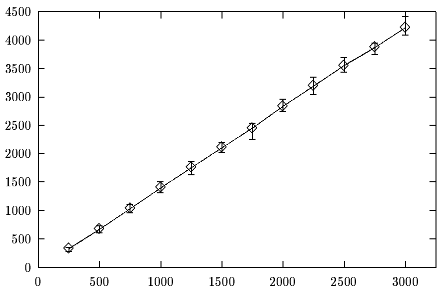
Figure 6.17: RandomMap

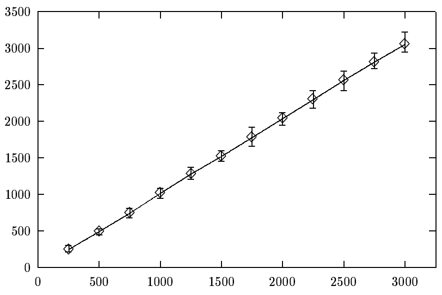
Figure 6.18: RandomRect

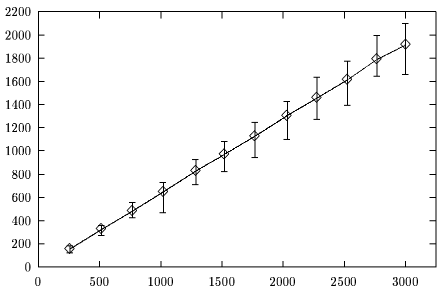
Figure 6.19: DenseMap

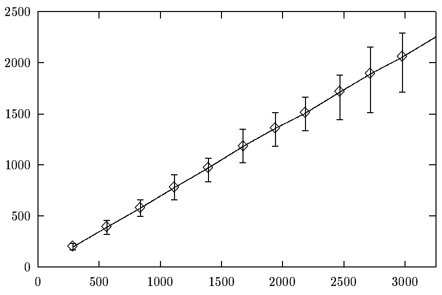
Figure 6.20: DenseRect

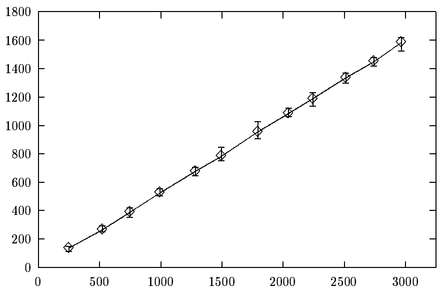
Figure 6.21: HardGrid

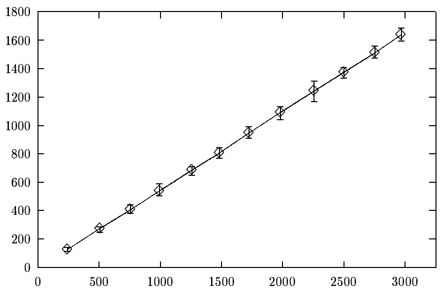
Figure 6.22: RegularGrid

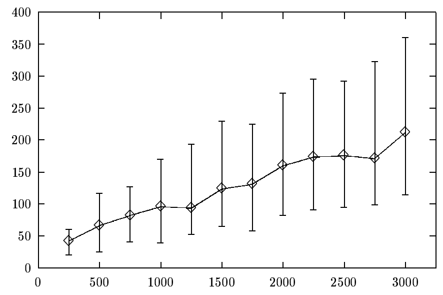
Figure 6.23: MunichDrillholes

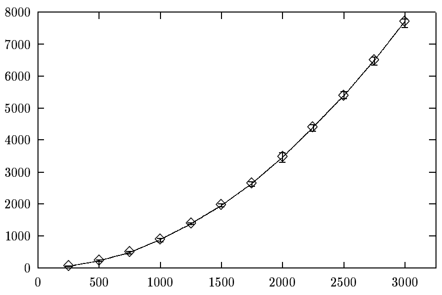
Figure 6.24: VariableDensity