SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper., **29**(15), 1355–1378 (1999)

Object-process Based Graphics Recognition Class Library: Principles and Applications

LIU WENYIN[1]* AND DOV DORI[2]
[1]*Microsoft Research–China, Sigma Center, No. 49 Zhichun Road, Beijing 100080, P.R. China*
(email: wyliu@microsoft.com)
[2]*Faculty of Industrial Engineering and Management, Technion–Israel Institute of Technology, Haifa 32000, Israel*
(email: dori@ie.technion.ac.il)

SUMMARY

We have developed a Graphics Class Library (GCL) for graphics recognition using the object-process methodology and object-oriented implementation. The purpose of the library is to supply generic code for graphics recognition algorithms to be used as ready made and easily extendible components in future systems. The library consists of reusable classes of graphic objects that appear in engineering drawings as well as in other classes of line drawings. A generic integrated graphics recognition algorithm is at the basis of the library, serving useful standard operations required for graphics recognition applications. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: software reuse; foundation class library; object-oriented design; object-process methodology; object-process diagrams; graphics recognition

INTRODUCTION

Being a domain of engineering, software engineering strives to establish standard, well-understood building blocks, which are expected to be developed and stored in libraries for common use and as a basis for extensions and modifications in the spirit of the reusability concept. Although software reuse had existed in software development processes since software engineering began as a research field in 1969, it was not before 1978 that the concept of reusability was clear in the minds of people as a solution to the 'software crisis' [1]. Currently, software reuse is getting a great deal of attention for its potential in increasing productivity, reducing costs, and improving software quality.

As a specialized form of software reuse, libraries of standard functions, such as mathematical subroutines, have been widely used in software development since the earliest applications of computers. As the software industry became stimulated by the advantages of software reuse, more domain-specific and powerful libraries have been developed in some domains, such as Matlab [2] for mathematics, and Khoros [3] for image processing. With the advent of the object-oriented software development approach, software reuse becomes easier to implement and libraries are constantly being extended to cover more common and

*Correspondence to: Liu Wenyin, Microsoft Research–China, Sigma Center, No. 49 Zhichun Road, Beijing 100080, P.R. China.

CCC 0038–0644/99/151355–24$17.50

Received 4 May 1999
Revised 13 August 1999
Accepted 13 August 1999

more complex data structures and operations, such as the Microsoft Foundation Class (MFC) library [4]. Two categories of reusable software components can be classified: horizontal and vertical [5]. Horizontal reuse refers to reuse across a broad range of applications areas, such as data structures, sorting algorithms, and user-interface mechanisms, such as MFC. Vertical reuse refers to components within a given application area that can be reused in similar applications with the same problem domain, such as the above-mentioned Khoros and Matlab packages. Although vertical reuse is less frequently employed, its reuse potential is not smaller than that of horizontal reuse.

The recognition of graphic objects from files of scanned paper drawings is a topic of increasing interest, known as engineering drawings interpretation, or document analysis and recognition. Although many algorithms and systems have been developed, the result is not satisfactory due to the complex syntax and semantics these drawings convey. To reduce the effort involved in developing basic algorithms for such systems, we have developed a Graphics Class Library (GCL) for graphics recognition for use as a framework in systems under development. This paper presents a vertical reusable software – the Graphics Class Library (GCL), which has been developed as part of the effort to develop the Machine Drawing Understanding System (MDUS) [6] for research in the domain of graphics recognition and engineering drawing understanding.

GCL has been developed using the Object-Process Methodology (OPM) [7–9] and applied in an object-oriented language, C++. Following the eight steps summarized by Cohen [10] for vertical software reuse development, domain analysis is carefully done by fully specifying requirements in the graphics recognition process before the design and implementation of the GCL. The result is a quality library of highly reusable and extendible classes and operations prevailing in the domain of graphics recognition.

The library consists of a variety of graphics classes, as well as auxiliary classes of the Graphics Database (GDB) for managing these graphic objects, Raster Image for original data reference, the Planar Position Index (PPI) for indexing these graphic objects, planar areas (Point, Rectangle, and Slanted Rectangle) for planar area operations, such as searching graphics within a given area, and the Viewer for displaying these graphic objects. The auxiliary classes provide for convenient and efficient storage, search, retrieval, and manipulation of graphic objects. Many complex operations (e.g. shape operations) that are useful in the graphics recognition process are developed and included in the GCL. A generic integrated graphics recognition algorithm, which is included in the GCL as a template function, is an important component of the library, as explained in the sequel.

THE OBJECT-PROCESS METHODOLOGY

The Object-Process Methodology (OPM) [7–9] is a system analysis and design approach that combines within a single modeling framework ideas from Object-Oriented Analysis (OOA) [11–13] and Data-Flow Diagrams (DFD) [14] to represent both the static/structural and dynamic/procedural aspects of a system in one coherent frame of reference. The use of a single model eliminates the integration problem and provides for clear understanding of the system under consideration. The Object-Process Diagram (OPD), whose symbol set is shown in Figure 1, is OPM's graphic representation of objects and processes in the universe of interest, along with the structural and procedural relationships that exist among them. Due to synergy, both the information content and expressive power of OPDs are greater than those of DFD and OOA diagrams combined. We proceed with a brief introduction of the OPM principles.

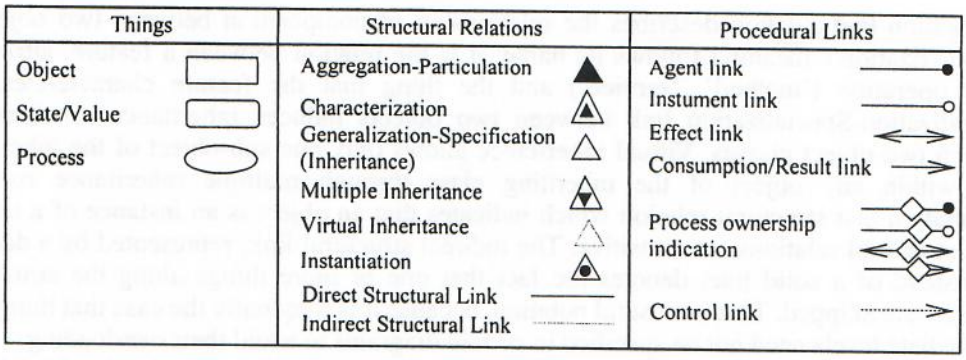


Figure 1. OPD symbol set

Things

In OPM, both objects and processes are treated analogously as two complementary classes of *things* – elementary units that make up the universe. An *object* is a persistent, unconditional thing. A *process* is a transient thing, whose existence depends upon the existence of at least one object. These terms were originally proposed for systems analysis in OPM [7]. From the design and implementation viewpoint, an object can be regarded as a variable with a specified data type, while a process is a function or a procedure operating on variables, which are objects.

An *object* class is a template of all objects that have the same set of features and behavior patterns, and whose corresponding name in the OO terminology is simply *class*. Similar to SmallTalk, an OPM object class can also be thought of as an object. This concept renders the class a relative term rather than absolute. It is relative with respect to the objects that are instantiated from it and provides for *instantiation hierarchy*. A *state* of a thing at a given point in time is the set (or vector) of attribute values the thing has at that point in time.

A very important feature of things (objects and processes) in OPDs is their recursive and selective scalability, which provides for complexity management through controlling the visibility and level of detail of the things in the system. In general, things are scaled up (zoomed in) as we proceed from analysis to design, and to implementation. The scaling capability provides for function definitions and calls. Specifying generalization-specialization among processes enable the establishment of inheritance relations among processes in a manner similar to inheritance among objects.

While OPM has been applied to system analysis and design [7,8], the expressive power of OPDs also makes them very instrumental in specifying the finest details of algorithms that are later implemented in some OO languages. The selective recursive scaling further facilitates the detailed design and algorithmic representation [9]. The resulting consistency of algorithm descriptions across the different phases of the software development process is highly desirable, and makes it amenable to computer aided software engineering.

Relations

The relationships among objects are described using structural links. Certain structural relations between two objects, namely Aggregation-Particulation, Characterization, and Generalization-Specialization, are collectively referred to as the *fundamental relations*.

Aggregation-Particulation describes the relationship of composition between two objects. Characterization's meaning follows its name: it is the relation between a feature, attribute or an operation ('method', 'service') and the thing that the feature characterizes. A Generalization-Specialization link between two objects induces inheritance relationship between two object classes. Virtual inheritance allows only one sub-object of the inherited class within any object of the inheriting class through multiple inheritance routes. Instantiation is a structural relation which indicates that an object is an instance of a class. Many structural relations are transitive. The indirect structural link, represented by a dotted line instead of a solid line, denotes the fact that one or more things along the structure hierarchy are skipped. This is a useful notation, because it is frequently the case that things at intermediate levels need not be specified in certain diagrams to avoid their overloading.

The relationships between objects and processes are described by procedural links, which are classified into effect, consumption, result, agent, and instrument links. *Agents* and *instruments* are enablers of processes. They exist before the process execution, and their state (set of attribute values) is not changed by the process execution. An *effect* link links an *affected object* to the affecting process. An affectee is an object whose state is changed by the process. A *consumed object* is an object that is consumed (and destroyed) by the process, and it no longer exists after the process execution. A *resulting object* is a new object constructed as a result of the process execution.

In current OO languages, processes, referred to as methods, belong to and are defined within some particular object class, so that the function can be called to handle an object (instance) of such a class or the class itself. In OPDs we call this object (when the process is called to handle it) or the class (when the process is called to handle the class itself) the *owner* of the process, and indicate it with a diamond symbol at the process end of the link (shown in Figure 1). Objects other than the owner and the resulting objects, that have procedural links with the process, can be considered as parameters of the process.

Control flow

OPDs use the top-down time line [8] and the data flow implied by the procedural links to define some of the control-flow sequencing. Cases in which the control does not flow from top down are marked by *control links* [9]. A control link links a process or a state of an object to a process to explicitly indicate the flow of control. Control links describe sequential and 'GOTO' control-flow mechanisms. They need not be used when the partial order of processes is clearly defined by the data-flow dependency.

The branching control mechanism is represented by *control objects*. The number of possible branches is decided by the number of *states* (possible values) that the control object may hold. For two possible values, the control object represents an IF-THEN-ELSE statement. If the number of possible values is more than two, it represents a SWITCH statement. The conditional branching control-flows converge at some point to end the branching. The control link does not determine the exact process sequence. The process order can be arbitrarily chosen, as long as it is compatible with the partial order specified by the data and control-flow in the OPD.

OPDs allow loops of both data and control-flow. In such a loop, a starting process should be explicitly specified by a control link in order to start the iteration. A binary (two-state) control object is involved. One state (referred to as the *exit state*) leads to an exit from the iteration and another (referred to as the *loop state*) leads to the continuation of the iteration. The control object is governed by the results of a *testing process*. The continuation of the iteration should

finally go back to the starting process. With the new definitions, OPD can distinguish two patterns of iteration: while-do and repeat-until. The while-do pattern is characterized by a starting process (possibly the testing process), followed by its resulting control object. Repeat-until is characterized by a control object whose loop state leads the control link back to the starting process. A 'FOR' iteration, as a special form of while-do patterns, can be recognized by finding an index indicator involved in the iteration.

As an exception to the general branching mechanisms, a special kind of branch that occurs inside an iteration may not have a joint end, since the conditional 'BREAK' and 'CONTINUE' control mechanisms are allowed in an iteration. A control link leading from the iterating process to the starting process is a 'CONTINUE' link, while a control link leading from the iterating process to the end of the iteration is a 'BREAK' link.

The details of a process (or *procedure* or *function*, as it may be called) are expressed in an OPD by scaling the process up. A special process, referred to as the *return process*, is introduced to terminate a procedure or function, as is done in many programming languages. We may express a recursion structure in OPDs by using the same process inside the blow-up of the process to express recursion. The same process may also occur more than once at the same level, if necessary, to process different objects. At least one control branch should be involved in the recursion to terminate it.

DOMAIN ANALYSIS

Graphics recognition is an important basic problem in engineering drawings interpretation, an area within the document analysis and recognition domain, interest in which is constantly increasing as more research into and development of experimental and commercial systems to solve this problem is conducted. Although algorithms related to this problem perform various functions, they share common knowledge and use identical building blocks. The Graphics Class Library (GCL) has been developed as a repository of algorithms in the domain of graphics recognition. The first step in building the library is domain analysis [15], in which the common knowledge is identified, captured, and organized. As an introduction to this domain analysis, we briefly describe the graphics recognition problem and the solution approach.

The graphics recognition problem

The engineering drawings interpretation accepts as input the raster image of a scanned drawing. Vectorization, or raster-to-vector conversion, applied on the raster image, yields coarse bars and polylines. Extra processing yields fine bars and polylines in complex and noisy drawings. After vectorization, the coarse bars and polylines are input to the extraction of text, arcs, dashed lines, and other forms of higher level graphic objects. We refer to this procedure as *graphics recognition*.

We define the problem of graphics recognition as grouping the raw wires resulting from the vectorization according to certain syntax rules and recognizing these groups as types of graphic objects and determining their attribute values. The low level graphic objects include bars, polylines, and arcs. Higher level graphic objects include characters and text, arrowheads and leaders, dashed lines, entities (geometric contours), hatched areas, and dimension sets.

Bars and polylines are relatively easy to detect while arcs are more difficult, due to their complex geometry. Although quite a few arc segmentation algorithms have been developed [16,17], the task still seems be to a tough problem. The Hough Transform (HT) can normally be used for arc segmentation in the case of isolated points that potentially lie on

circles or circular arcs [16]. However, its high complexity in both time and space makes such arc segmentation algorithms less practical for engineering drawings. Perpendicular Bisector Tracing (PBT) [17] is the first vector-based method of arc segmentation. Since PBT examines only the bar fragments output by the vectorization process, it is efficient in terms of both time and space. All arc segmentation algorithms discussed above are designed to segment only solid arcs. Research on the detection of arcs of other styles is rare. Arc segmentation can also be done using patterns and clues that the line segments, vectorized from an arc, constitute a chain of pseudo line segments that is shorter than some statistical threshold, and which is delimited between two long straight lines [18].

Line style detection has also been studied by several groups [18–23], but it is treated only as a small, side issue in these works. A pixel-based method [19] usually employs the HT to detect dashed circles and dashed straight lines in several steps. This pixel-based method segments one class of dashed lines in each step, and it is computationally expensive. A semi-vector-based method [20] is used to find dashed segments which have special graph structures. Other groups use vector-based algorithms to detect discontinuous lines. The Celesstin [21] system can detect both dashed lines and dash-dotted lines according to the French Standard NF E 04-103. ANON [22] deals with finding dashed lines in engineering drawings by looking for chains of short lines, which are also used as clues by other methods [18,23].

Text, as a special graphics object in engineering drawings, requires different processing. First, the character image should be segmented from the drawing using a procedure called *text segmentation* (or *text/graphic separation*), which we include as part of the graphics recognition problem. The character image is then input to an Optical Character Recognition (OCR) sub-system for recognition. Text segmentation can be done either at the raster level before vectorization [24,25] or at the vector level, i.e. after vectorization [26]. While OCR problems for clean printed and neat hand-printed characters has almost been solved, text segmentation in engineering drawings is still an open problem due to inherent difficulties, such as the text/graphics mixture and connectivity, variation in character location, size and orientation, handwritten characters, and noise.

Projected to 2D views, a 3D solid object becomes a set of connected empty blocks (possibly hatched, when a section is shown) with thick lines as their contours. These blocks are also called 2D graphic entities. The thick line contours describe the shape of the objects. To precisely express other information (e.g. measurements) of the objects, some annotations (namely, dimensioning) are also added to the drawing. The drawing is therefore a mixture of two types of representations: the first is the projected shape of the actual object (part or assembly); and the second, superimposed on the first, is the annotation (dimensions) – formal, standard-based language that expresses other attributes of the objects. We also include the recognitions of the entities and the dimensions in the graphics recognition. Methods of dimension recognition have been investigated and found to be successful to some extent for some particular standards on limited experimental data [27,28]. The dimension-set recognition usually starts from arrowhead detection on image drawings. The extraction of leaders and witness lines then follows. Finally, the detected text blocks are associated with the dimension lines. ANON [22] does detection of both physical outlines and dimensions using a low level image analysis. The recognition of the image part is confined to the detection of entities, which are empty or hatched closed geometric contours made of thick lines. Celesstin [21] deals with block detection in their Cellestin system. Minimal closed polygons with thick lines are detected as blocks.

In spite of the existence of the algorithms and systems reported above, no research report has yet proposed the detection of all classes of graphic objects in a generic, unifying

algorithm. As of now, each class of graphic objects requires a particular recognition algorithm. Moreover, in the process of recognizing each class of graphic object, almost all methods cluster all the potential constituent line segments at once, while their type and attributes are determined later. This blind search procedure tends to introduce inaccuracies in the grouping of the graphic primitive components that constitute the graphic objects, which ultimately account for inaccurate graphics recognition. Our approach is more flexible and adaptive, as it constantly checks the syntactic and semantic constraints of the graphic object while grouping its primitive components.

Domain analysis of graphics recognition

Since the Graphics Class Library (GCL) is aimed at being a vertical reusable software component in graphics recognition, we carry out the domain analysis of graphics recognition for building the GCL following the methodology summarized by Cohen [10] in the following steps.

Select specific functions/objects

The graphic objects that appear in engineering drawings include solid straight lines, solid arcs, solid polylines, dashed straight lines, dashed arcs, dashed polylines, dash-dotted straight lines, dash-dotted polylines, dash-dot-dotted straight lines, dash-dot-dotted arcs, dash-dot-dotted polylines, character boxes, character string boxes, logic text boxes, filled arrowheads, hollow arrowheads, straight leaders, angular leaders, entities (close geometric contours), hatched areas, longitudinal dimension sets, angular dimension sets, radial dimension sets, diametric dimension sets, etc. These graphic objects are selected to be included in the GCL for reuse. Other graphic objects that appear less frequently in engineering drawings may be considered for inclusion in later versions of the GCL, and some of them may be derived from the current contents of the GCL.

The main reusable function is the recognition processes of these graphic objects. It is the core of the GCL. The common and frequently used behavior of these graphic objects, which includes naming, credibility testing, displaying, moving, rotating, management in the graphics database (such as insertion and retrieval), geometric computation, and standardized file I/O [29,30], are included in the GCL, as they are also frequently needed in graphics recognition systems.

Searching for graphic objects within a particular given area in a drawing is a very common operation within the graphics recognition process. Traditional graphics recognition algorithms perform this kind of search by applying a sequential search throughout the entire graphics database, and testing each graphic object if it passes through the given area. This brute-force, straightforward search method is of ($O(N)$) time complexity, where N is the total number of objects in the database. Since this search function is frequently used, we select, efficiently implement and include it in the GCL for reuse with the area parameter being a point, a rectangle, or a slanted rectangle.

The raster image of the drawing is the input of the graphics recognition process, and the pixel operations on it are frequently used. It is selected as a reusable component in the GCL. The graphics database is an important object in the graphics recognition process. It is therefore selected and included in the GCL, along with its management functions, such as efficient storage, query, and retrieval of graphic objects.

Abstract functions/objects

We use some graphics classes to abstract all the graphic objects. These graphics classes are as follows, whose meanings follow their names: Bar (Solid Straight Line), Arc (Solid Arc), Polyline (Solid Polyline), Dashed Straight Line, Dashed Arc, Dashed Polyline, Dash-Dotted Straight Line, Dash-Dotted Polyline, Dash-Dot-Dotted Straight Line, Dash-Dot-Dotted Arc, Dash-Dot-Dotted Polyline, Charbox (Character Box), Stringbox (Character String Box), Logic Text Box, Filled Arrowhead, Hollow Arrowhead, Bar Leader (Straight Leader), Arc Leader (Angular Leader), Entity, Hatched Area, Longitudinal Dimension Set, Angular Dimension Set, Radial Dimension Set, and Diametric Dimension Set. Each class abstracts the common structures and behaviors of the objects of such class.

The area search function is abstracted as a function that realizes the search for graphic objects from the graphics database within a particular area, which may be denoted by a point, a rectangle, or a slanted rectangle. To facilitate the area search, we have introduced the *position index* data structure [31], which indexes the graphic objects using their planar positions. The class Planar Position Index (PPI) is used to abstract such a data structure as part of the GCL. Point, Rectangle, and Slanted Rectangle, which abstract points, rectangles and slanted rectangles, respectively, are auxiliary classes for the PPI search mechanism.

The class Raster Image is used to abstract the raster image and the common operations on it. The Graphics Database (GDB) class is abstracted for organizing and managing the graphic objects, which contains these graphic objects and support for ease of storage, query, retrieval and manipulation of them. The class Viewer represents all the information needed to display the graphic objects in a window. Each particular graphics class displays its objects using the information transferred by the Viewer.

Define taxonomy

As noted before, the objects and classes are categorized into graphics classes and auxiliary classes. The graphic classes, in turn, can be classified into line classes, text classes, entity classes, and annotation classes (arrowhead, leader, and dimension set classes).

Lines are classified into types by two attributes: shape and continuity. The three line shape classes are straight, circular, and polygonal. The four line style classes are solid, dashed, dash-dotted, and dash-dot-dotted. Lines are classified by the continuity attributes into Continuous (Solid) Line classes and Discontinuous Line class. The three solid line classes are Bar, Arc, and Polyline, whose objects have the solid line style. The discontinuous line class includes all lines that have a non-solid line style.

The text classes are Charbox (Character Box), Stringbox (Character String Box), and Logic Text Box. The entity classes are Entity and Hatched Area. The annotation classes are Arrowhead, classified into Filled Arrowhead and Hollow Arrowhead, Leader, classified into Bar Leader and Arc Leader, and Dimension Set (including all dimension set classes). The auxiliary classes are Graphics Database (GDB), Planar Position Index (PPI), Point, Rectangle, and Slanted Rectangle, and Viewer.

The functions of the graphic objects are classified as recognition, naming, credibility testing, displaying, moving, rotating, geometric computation, standardized file I/O [29,30], and graphics database management functions, such as inserting, position indexing [31], and retrieval. The functions are classified into two groups: the first includes those functions that characterize a group of classes, and their process details are identical for each class; the second the functions that, while characterizing a group of classes, each has different implementation details (behavior) for the various classes.

Identify common features

As the names of the graphic objects indicate, genericity of both structure and behavior exists within graphics classes.

The common structural feature is that each graphic object is composed of a group of graphic primitive components constrained by certain syntactic rules of its class. Thus, for example, a dashed line is composed of a set of dashes constrained by a line geometry and a dashed pattern; a character is composed of a set of strokes close enough to each other within a limited area; and a dimension-set consists of text, one or two leaders, and references.

The most common behavior feature shared by the graphics classes is that their (vector-based) recognition processes follow a common framework. The underlying mechanism of this common framework is a stepwise recovery of their multiple components that obey certain syntactic rules. Rather than finding all the vector components of the graphic object at the same time, as is done in most current graphics recognition algorithms, we find for the graphic object being detected only one new component that best meets the conditions constrained by its corresponding rules. Before searching for the next component, we update the current graphics attribute values. In this way, the current graphic is detected to the highest extent possible, while avoiding many recognition false alarms. For example, in dimension set detection, the first key component is a textbox. The extension area is the rectangle obtained by enlarging the textbox to its four sides by half of its height. The closest leader found in this area is combined with the textbox and a dimension set is found.

Other processes of these graphic objects, such as displaying, moving, and area indexing, also follow generic patterns within groups of graphics classes. Organizing them in an inheritance hierarchy is not only possible, but also necessary, for code efficiency and reusability. For example, both the solid arc class and the dashed arc class inherit from the arc class, and a dashed arc inherits from both the arc class and the dashed line class.

Identify specific relationships

Examining the objects involved in graphics recognition, we find that the main relationships among them are the relationship between the graphics database and the graphic objects, and the relationship between the graphics database and the position index that helps index the graphic objects by their planar positions. The most important relations are the roles of these objects in the graphics recognition process.

These relationships should be well analyzed and designed in appropriate forms in the GCL so that they depict the relationship models in the real world, and make the entire GCL highly reusable and extendible.

Abstract specific relationships

Following both OOA and OPA [7], the relationship between a graphics class and its objects is Instantiation. The relationship between the graphic database and the graphic objects is shown in Figure 2 as aggregation-particulation [7]. The graphic database consists of a set of lists, each of which contains a particular class of graphic objects.

The position index is used to index and help search for graphic objects in the graphics database. Neighborhood objects like Point, Rectangle, and Slanted Rectangle are used as arguments for area search functions. Figure 3 depicts the procedural relationship among these things.

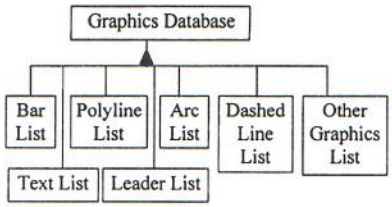


Figure 2. The OPD depicting the relationships between the graphics database and the graphics objects lists

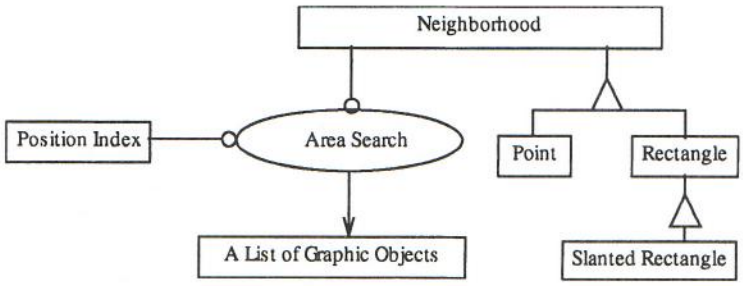


Figure 3. The OPD depicting the relationship among the Area Search process, the position index, a neighborhood, and a list of discovered graphic objects

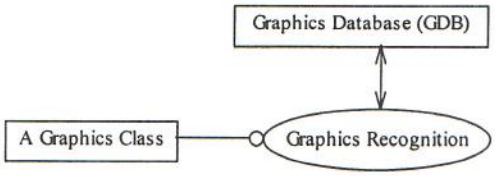


Figure 4. The OPD depicting the relationship among the graphics recognition process, the graphics database, and a particular graphics class

Derive a functional model

The graphics recognition process consists of taking a particular graphics class and the currently available graphic objects in the graphics database as input, outputting a set of detected objects of the class, and inserting them into the graphics database. Figure 4 depicts the relationships among the graphics recognition function, the graphics database, and the graphics classes.

Define a domain language

The main classes and terms used in our domain are defined below.

1. *Primitive* – a generic name for a graphic object that appears in an engineering drawing.
2. *Line* – a generic name of an abstract class of graphic objects in line drawings, each of which is the trace of a non-zero width pen that moves from a start point to an end

point, follows a certain trajectory, which may be constrained by a geometric function and optionally leaves invisible segments according to some pattern. The width of the pen is called the *line width*. The start point and the end point are called the *endpoints*. The trajectory is called the *line medial axis*. The geometric form of the line's medial axis is called the *line shape*. The alternation of visible and invisible segments, as determined by their lengths and sequence pattern, is called the *line style*. The visible and invisible segments are called *dashes* and *gaps*, respectively.

We only consider *simple* lines, i.e. lines that do not intersect themselves. All lines share the following common attributes.

(a) A line has two endpoints, which limit the extent of the line. Circles and polygons may also be considered as lines whose two endpoints coincide.
(b) A line has a unique, ideally constant, non-zero *width* between the two endpoints.
(c) A line is characterized by the *style* attribute, whose values, explained in Definitions (3)–(7), are solid, dashed, dash-dotted, or dash-dot-dotted.
(d) A line is characterized by the *shape* attribute, whose values are straight, circular arc, or polygonal, as listed in Definitions (8)–(10).

3. *Solid Line* – a line whose style is solid, which means that the entire line is continuously visible and traceable from end to end. In other words, it consist of a single dash and no gap between the two endpoints.
4. *Discontinuous Line* – a line whose style is not solid, i.e. it consists of at least two dashes separated by one gap.
5. *Dashed Line* – a discontinuous line whose dashes are relatively equal and long, and whose gaps are relatively equal and short.
6. *Dash-dotted Line* – a discontinuous line whose dashes can alternately be distinctly classified as long and short. The short dashes are called *dots*. The dashes within each group are of relatively equal lengths. At least in hand-made drawings, the two dashes at the two line ends are usually long.
7. *Dash-dot-dotted Line* – a discontinuous line similar to the dash-dotted line, except that every dot of the dash-dotted line is replaced with a dot-gap-dot pattern (two neighboring dots with a gap between them).
8. *Straight Line* – a line whose medial axis is constrained by equation (1):

$$ax + by = c \tag{1}$$

where (x, y) is the coordinate pair of any point on the line's medial axis, while a, b and c are parameters. The line is limited by two endpoints $p_0(x_0, y_0)$ and $p_1(x_1, y_1)$.

9. *Circular Line* – a line whose medial axis is constrained by equation (2):

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \tag{2}$$

where (x, y) is any point on the line's medial axis, while (x_c, y_c) is the circular center and r is the circular radius. The line is limited by the two endpoints $p_0(x_0, y_0)$ and $p_1(x_1, y_1)$, going counterclockwise from p_0 to p_1. The two endpoints may coincide when the circular line is a full circle.

10. *Polygonal Line* – a line whose medial axis is represented by a sequence of N characteristic points p_i, where $i = 0, 1, \ldots, N - 1$. The medial axis segment between every two neighboring characteristic points p_i and p_{i+1} is constrained by equation (3):

$$a_i x + b_i y = c_i \quad i = 0, 1, \ldots, N - 2 \tag{3}$$

where (x, y) is the coordinate pair of any point on the line's medial axis between two points $p_i(x_i, y_i)$ and $p_{i+1}(x_{i+1}, y_{i+1})$, while a_i, b_i and c_i are parameters. The entire polygonal line is limited by the two endpoints $p_0(x_0, y_0)$ and $p_{N-1}(x_{N-1}, y_{N-1})$. The polygonal line may be a closed polygon, whose characteristic start and end points coincide.

A polygonal line is usually composed of a sequence of solid, equal-width lines linked end-to-end with optional intermediate gaps. It may also be used to approximate all line shapes other than straight and circular arc forms, including some high order and free form curves.

Combining the four line styles and three line shapes, we obtain the following 12 line classes which are listed in Definitions (11)–(22).

11. *Bar* – a solid straight line.
12. *Arc* – a solid circular line.
13. *Polyline* – a solid polygonal line, consisting of a chain of equal-width bars linked end-to-end.
14. *Dashed Straight Line* – a line whose style is dashed and whose shape is straight.
15. *Dashed Arc* – a line whose style is dashed and whose shape is circular.
16. *Dashed Polyline* – a line whose style is dashed and whose shape is polygonal.
17. *Dash-dotted Straight Line* – a line whose style is dash-dotted and whose shape is straight.
18. *Dash-dotted Arc* – a line whose style is dash-dotted and whose shape is circular.
19. *Dash-dotted Polyline* – a line whose style is dash-dotted and whose shape is polygonal.
20. *Dash-dot-dotted Straight Line* – a line whose style is dash-dot-dotted and whose shape is straight.
21. *Dash-dot-dotted Arc* – a line whose style is dash-dot-dotted and whose shape is circular.
22. *Dash-dot-dotted Polyline* – a line whose style is dash-dot-dotted and whose shape is polygonal.
23. *Text* – a graphic object whose components are digits, characters, spaces, punctuation marks, and special symbols.
24. *Textbox* – a minimal rectangle that encloses a particular text without any non-text element, and any part of other text, and whose slant corresponds with the orientation of the text. The textbox of a text is the union of the charboxes (defined below) of its character components.
25. *Charbox* – character box, a box which bounds a single character.
26. *Stringbox* – character string box, a textbox of a single string of characters.
27. *Logical Textbox* – a textbox, in which all elements are logically connected and refer to a common element of an engineering drawing. One logical textbox can contain from one to three string textboxes. A logical textbox is usually used for a dimensioning text, which consists of a single string textbox as the nominal dimension and another one or two string textboxes as its tolerance (e.g. $5^{\pm 0.5}$).
28. *Entity* – an area whose boundaries are solid thick lines of any geometry. It represents the projected face of an object. The circumference of the Entity is usually closed. When it is not closed, the open part may be either missing or a free hand drawn thin line.
29. *Hatched Area* – an entity within which there is a group of slant parallel thin bars representing a cross section of a 3D object.
30. *Arrowhead* – an equilateral triangle shape of graphic object. The point formed by the two equal edges is called the *tip* and the edge opposite the tip is called the *back* of the arrowhead. A tail is always attached to the arrowhead's back to form a leader.

31. *Hollow Arrowhead* – a arrowhead whose area is empty.
32. *Filled Arrowhead* – a arrowhead whose area is filled with the color of the boundary.
33. *Leader* – a combination of an arrowhead of any type and a tail, which is a solid straight or circular line attached to the back of the arrowhead.
34. *Double Leader* – a leader which has two arrowheads, one at each end of the solid line such that it is used twice, each time as the tail of one of the arrowheads.
35. *Bar Leader* – a leader whose tail is a bar.
36. *Arc Leader* – a leader whose tail is an arc.
37. *Dimension Set* – a set of a dimensioning textbox, a leader set (one or two leaders), and optionally, a *guidance* (an extension of the leader tail from the arrowhead tip to outside) and two *references* (a line at which the arrowhead points)
38. *Longitudinal Dimension Set* – a dimension set containing two bar leaders and two bar references.
39. *Radial Dimension Set* – a dimension set containing one bar leader and one arc reference.
40. *Diametric Dimension Set* – a dimension set containing two bar leaders and two arc references.
41. *Angular Dimension Set* – a dimension set whose leaders are arc leaders two bar references.
42. *Graphics Database* – a data structure that servers as a repository of graphic objects.
43. *Point* – a planar point characterized by a pair of x and y coordinates.
44. *Rectangle* – a four-side polygon whose sides are either horizontal or vertical. It is characterized by a left-top point and a right-bottom point.
45. *Slanted Rectangle* – a rectangle which is rotated by any angle.
46. *Planar Position Index* – a data structure that indexes the graphic objects using their planar positions in the drawing [31].
47. *Viewer* – a data structure that contains the entire package of system information useful when displaying a graphic object, such as the window handle and the device content or the graphics content.
48. *Area Search* – a search for graphic objects from the graphics database within a particular area, which may be denoted by either a point, a rectangle, or a slanted rectangle.
49. *First Key Component* – a vector component of a graphic object that best differentiates the object class from other object classes. It is used as a clue of the possible existence of the object.
50. *Stepwise Recovery* – a procedure that discovers the vector components of a graphic object one at a time.
51. *Extension* – a procedure that applies the stepwise recovery of the components of a graphic objects.

DESIGN

Based on the domain analysis, we design the Graphics Class Library (GCL) using the object-process methodology. The GCL consists of the graphics classes organized in an inheritance hierarchy, the generic graphics recognition algorithm, some auxiliary classes used in the graphics recognition process, and some frequently used operations.

Design of the Graphics Classes Inheritance hierarchy

We use the definitions of the graphics classes to construct the graphics classes hierarchy, which is shown in Figure 5, where the abstract class Primitive is at the top of the hierarchy.

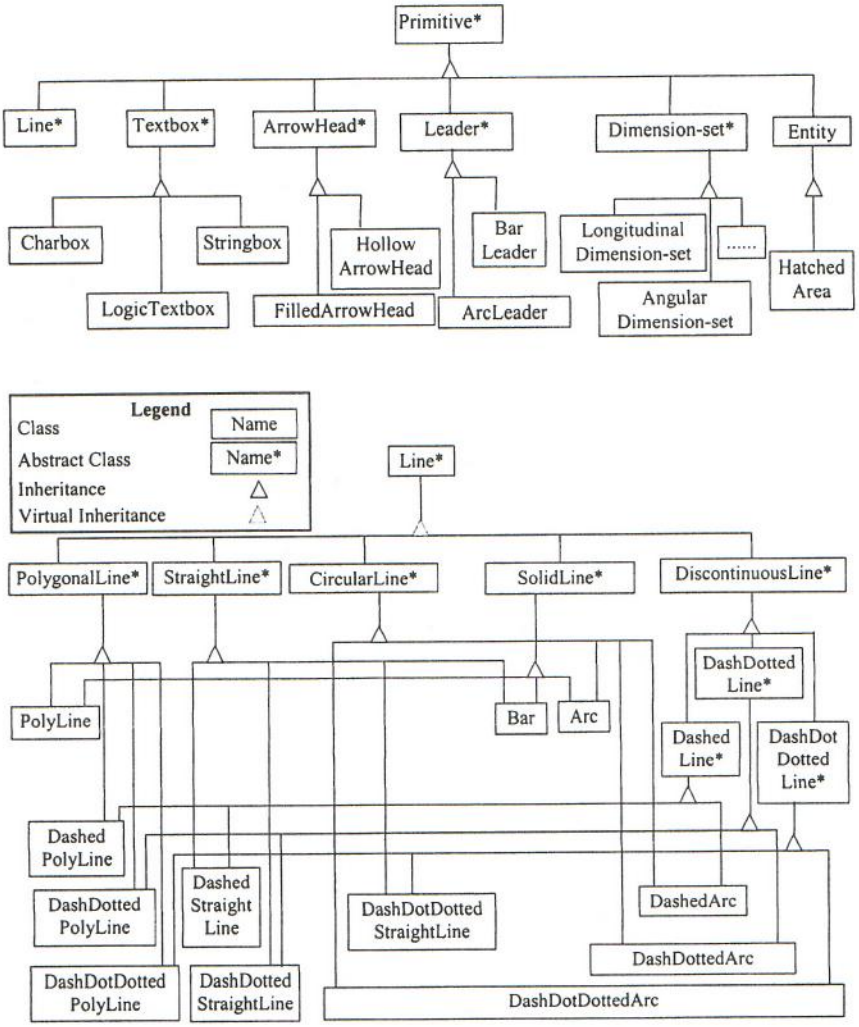


Figure 5. OPD of the inheritance hierarchy of the classes of line objects

At the second level of the hierarchy there are several abstract classes. The class Line abstracts the most common features of all line classes. Textbox is inserted into the hierarchy to generalize the features of the classes Charbox, Stringbox, and Logic Textbox. Arrowhead generalizes the classes Filled Arrowhead and Hollow Arrowhead. Leader generalizes the classes Bar Leader and Arc Leader. Both Entity and Hatched Area are concrete classes, but we define Hatched Area as a subclasses of Entity. Classes of Longitudinal Dimension Set, Angular Dimension Set, and other kinds of dimension set are generalized as Dimension Set.

Line is an abstract class because the style and shape are not specified as its attributes. Each shape of line is a lower level abstract line class, whose shape is specified but the style is not. There are three such classes: Straight Line, Circular Line, and Polygonal Line. Each line style is also represented by an abstract line class, with the line style specified and the line shape unspecified. These classes are Solid Line, Dashed Line, Dash-dotted Line, and Dash-dot-

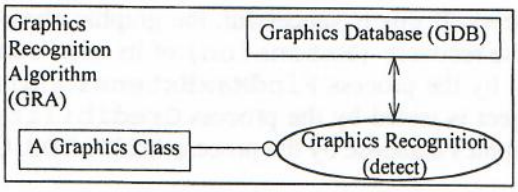


Figure 6. The OPD the graphics recognition (detect) process of detail design and implementation phase

dotted Line. An abstract class named Discontinuous Line is also inserted into the hierarchy as an abstraction of the three line classes that have gaps in their objects, i.e. Dashed Line, Dash-dotted Line, and Dash-dot-dotted Line. Finally, the 12 concrete line classes are located at the bottom of the hierarchy. The line attributes in each concrete class are fully specified through multiple inheritance from two abstract classes, one specifying the line shape and the other specifying the line style.

To avoid the inheritance of two copies of the Line object by each of the 12 concrete classes, as normally happens in a multiple inheritance hierarchy – one through the line shape class and the other through the line style class – we implement virtual inheritance, symbolized by the dotted triangle between Line and each one of its immediate specifications (inheriting classes).

Design of the Generic Graphics Recognition algorithm

Since the graphics recognition process is modeled generically, as shown in the OPD of Figure 4, we design it to be a template (parameterized) function, which takes a graphics class as a parameter for initiation, and the graphics database as input and output parameters, as shown in Figure 6.

The following line of C++ code gives the interface of the template function:

```
template <class AGraphicsClass> void detect(AGraphicsClass*, GraphicDataBase&gdb);
```

The parameterized design provides for easy reuse of the code, even within the GCL, since only a small piece of code is used for the recognition process of all graphics classes. The first parameter in this function requires a type of `AGraphicsClass*` for instantiation for this class; its actual value is not used in the function body. The second parameter `gdb` is used for both input and output.

In the detailed design phase, the OPD of Figure 6 is elaborated into many OPDs [9], such as that in Figure 7, which presents more details of the graphics recognition process. As shown in Figure 7, the template function `detect()` is exploded to show its algorithmic details regarding the generic graphics recognition algorithm [6]. It consists of two steps based on the hypothesis-and-test paradigm. The first step is hypothesis generation, in which we assume the existence of a graphic object of the class being detected by finding its first key component. This is implemented by `FindKeyComponent()`, as shown in Figure 7(a). The second step is the hypothesis test, in which we prove the presence of such a graphic object by constructing it from its first key component and its other components that are detected serially. This is implemented by the template function `Construction()` in Figure 7(a), which, in turn, is blown up to show its algorithmic details in Figure 7(b). Here, an empty Graphic Object of the Graphic Class is first created by the 'new' process. It is then filled with the Key Component object found by the `FindKeyComponent` process and transferred

into the `FillWith` process. If this is successful, the graphic object is further extended as far as possible by stepwise recovery (`Extension`) of its other components in all possible directions, as determined by the process `FindMaxExtensionDirections`. After that, the extended graphic object is tested by the process `CredibilityTest`. If it passes the test, it is added to the graphics database by the process `AddToGDB`. Otherwise it is deleted.

Design of the member functions of the graphics classes

As shown in Figure 7, the following member functions are involved in the graphics recognition process. `FindKeyComponent` is owned by the particular graphics class, and is therefore designed as a static (class) function for every concrete graphics class. Others, `FillWith`, `Extension` `FindMaxExtensionDirections`, `CredibilityTest`, `AddToGDB`, are owned by a particular graphic object, and are therefore defined as regular member functions.

Since every particular graphics class has such member functions, they can be abstracted to appropriate classes as part of the genericity of a certain group of classes. The functions of a group of classes that have exactly the same process details are abstracted and defined as regular member functions of the class that abstracts the group of classes. Some functions of a group of classes may not have exactly the same process details. They are defined as virtual member functions of these graphics classes and their base class. If the base class does not 'know' the function details, the function is defined as a pure virtual function of the base class. For example, the functions `FillWith`, `Extension` `FindMaxExtensionDirections`, `CredibilityTest`, and `AddToGDB` prevail in all graphic classes. Hence, they can be abstracted within the class Primitive, since it is at the top of the inheritance hierarchy. They are defined as pure virtual member functions of the class Primitive because they cannot be implemented in it.

The Line class is characterized by many virtual functions, such as the function that retrieves the endpoints of a line object. However, the class Line does not know how to retrieve the endpoints of a line object because it does not know the exact line geometry. Therefore, this function is defined as a pure virtual member function of class Line. Nevertheless, we know that any line object has two extension directions, outward from each endpoints. Hence, the function `FindMaxExtensionDirections` is defined, and can be fully implemented in class Line; it returns the value 2. Moreover, the functions of getting the extending area, extending candidates, and the extension function also have the same algorithms; they are therefore defined and fully implemented in the class Line.

Design of auxiliary classes

The class Graphics Database is designed to comprise lists of graphic objects. We also define the Planar Position Index (PPI) as a part of the graphics database, as shown in the OPD of Figure 8. Thus, every graphic object has two references in the graphic database: one from the categorized sequential lists, which facilitates category and sequential search and retrieval of the graphic objects from the graphic database; and one from the position index, which facilitates area or position search and retrieval of the graphic objects from the graphics database. This helps manipulate the graphics database and the graphic objects within it efficiently and effectively.

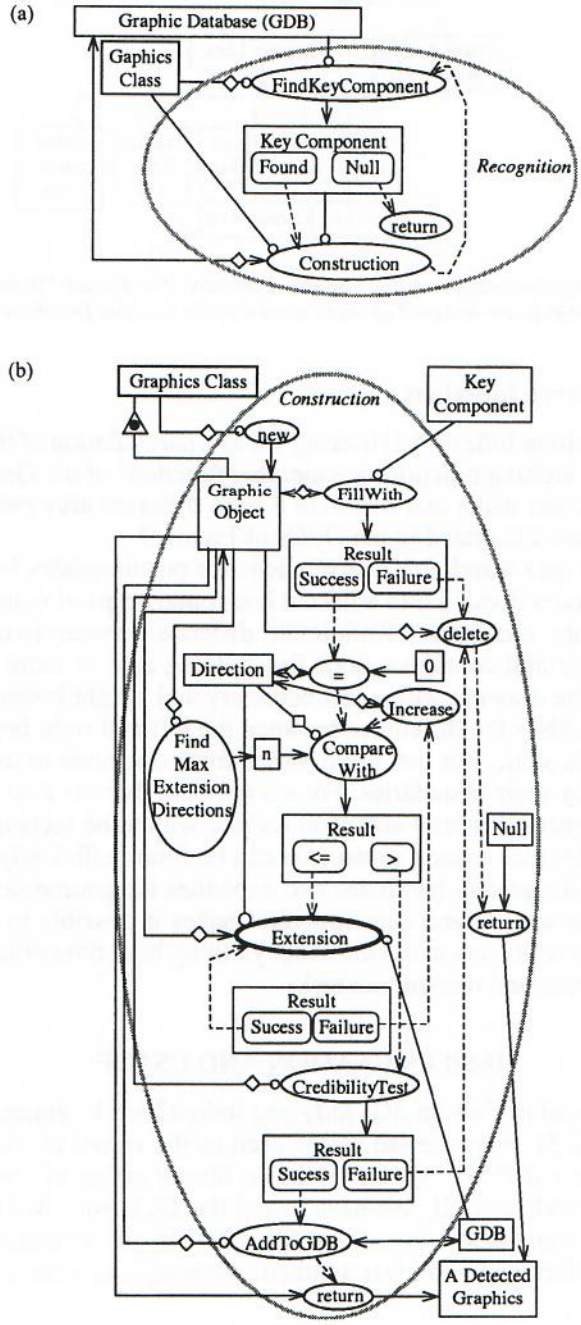


Figure 7. OPD illustration of the GRA (Process). (a) Explosion of 'detect' process in Figure 6. (b) Explosion of the 'construction' process in (a)

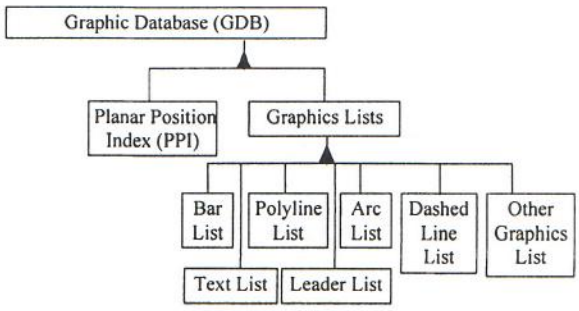


Figure 8. The OPD showing the structure of the Graphic Database. The Planar Position Index and the graphic objects are designed as particulation of the Graphic Database

Design of the area search functions

Since the Planar Position Index (PPI) is designed as a particulation of the Graphic Database, the interface of area searching functions are member functions of the Graphic Database class. We use the same function name and overload it with different area parameters. Three such area search functions are illustrated in the OPDs of Figure 9.

In order to make the area search function efficient, the position index is designed as follows. The entire drawing area is divided into adjacent horizontal strips of equal width, the value of which depends upon the parameter of minimum distance between two parallel lines. Each strip has a strip number and contains a node list, holding zero or more nodes. Each node is a rectangular area in the drawing with a left boundary and a right boundary. Its height is the strip width, and its width is the difference between the left and right boundaries. The height of all nodes is thus the same, but the width varies from one node to another. The nodes in each strip are sorted by their boundaries. For every node, there is also a pointer that points to the set of objects covering at least one pixel located within the rectangular node area. Due to the sorting, the nodes that cover a given area can be found efficiently using binary search in logarithmic time. This greatly facilitates and expedites the geometric objects' search in a given area. Hence, this specialized data structure makes it possible to realize the mapping from positions to graphic objects in the drawing, yielding high time efficiency of a variety of higher level segmentation and recognition tasks.

IMPLEMENATION AND USAGE

The GCL is implemented in C++ on SGI Indy and Indigo2 workstations (IRIX5.3) and SUN Sparcstations (Solaris2.5). It was tested and is used as the kernel of the Machine Drawings Understanding System (MDUS) [6]. The linkable library codes of these two versions are available from an ftp address [32]. We have tested the GCL with MDUS using real world drawings of various complexity levels. As we show in the experiments, the algorithm demonstrates high performance on clear synthetic drawings, as well as on noisy, complex, real world drawings.

The implementation of the generic graphics recognition algorithm

Following the design of the generic graphics recognition algorithm, we implement it using C++, and the implementation is shown in the C++ code in Figure 10.

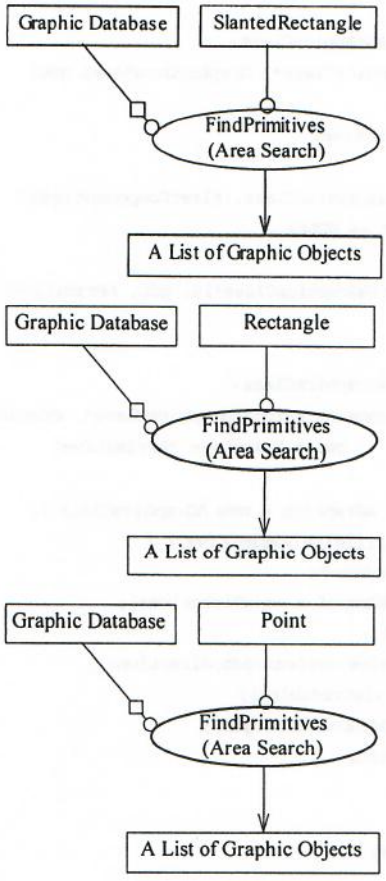


Figure 9. The OPD depicting the relationship among the area search function, the position index, a given area, and a list of discovered graphic objects

The user manual of the Graphics Class Library

The GCL can be easily extended to include newly defined classes. The only work the user needs to do is inheriting the new class from a proper one in the GCL and overriding necessary member functions. For example, if we want to add the class 'Symbol' into the GCL, it can be defined as in Figure 11.

To use the GCL in a system, the header files should be included in the code of the system and the GCL linkable library code should be linked to code in the project file.

A simple example of using the GCL requires only a `main.c++` file in the project file that defines the window interface, includes all the header files of GCL, and includes GCL code as a linkable library. In the `main.c++` file, the recognition process of a concrete graphic object class is triggered by an initiation of the template function `detect()`. For example, to detect dashed lines, we call the function in the following way:

```
detect((DashedLine*)0, aGraphicDataBase);
```

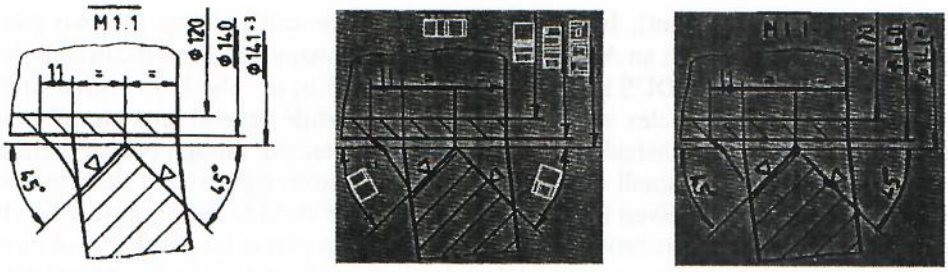
We have used the GCL within MDUS in the recognition of a variety classes of graphic objects [33–37]. Experiments and performance evaluation [36,37] show that it is successful


```
template <class AGraphicsClass>
void detect(AGraphicsClass*, GraphicDataBase& gdb)
{
 Primitive* APrimitive;
 do {
  APrimitive = AGraphicsClass::firstComponent(gdb);
  if (APrimitive == NULL)
    return;
  constructFrom((AGraphicsClass*)0, gdb, APrimitive);
 } while (1);
}
template <class AGraphicsClass>
AGraphicsClass* constructFrom(AGraphicsClass*, GraphicDataBase& gdb,
                    const Primitive *APrimitive)
{
 AGraphicsClass* AGraphics = new AGraphicsClass();
 if (AGraphics->fillWith(APrimitive)) {
  for (int direction=0;
    direction<=AGraphics->maxDirection();
    direction++)
   while (AGraphics->extend(gdb,direction));
  if (AGraphics->isCredible())
   AGraphics->addToDataBase(gdb);
   return AGraphics;
  }
 }
 delete AGraphics;
 return NULL;
}
```

Figure 10. Outline of the C++ implementation of the generic graphics recognition algorithm

```
class Symbol: public Primitive
{
 ...
 static Primitive* firstComponent(const GraphicDataBase& gdb);
 BOOL fillWith(const Primitive* Aprimitive);
 int maxDirection(void) const;
 BOOL extend(const GraphicDataBase& gdb, int direction);
 BOOL isCredible(void) const;
 void addToDataBase(GraphicDataBase& gdb) const;
 ...
};
```

Figure 11. Exemplified definition of class 'Symbol'



(a) (b) (c)

Figure 12. Graphics recognition by the GCL within MDUS: (a) original image, (b) recognized bars, texts, and leaders, (c) detected hatching lines within two groups

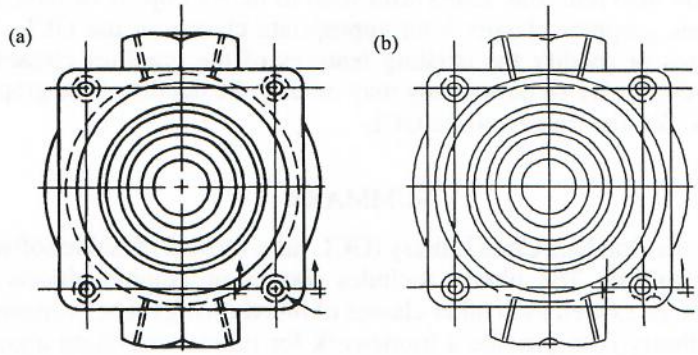


Figure 13. Graphics recognition by the GCL within MDUS: (a) original image, (b) detected circles, arcs, and dashed (including dash-dotted) (straight and circular) lines displaying in solid and one-pixel line widths

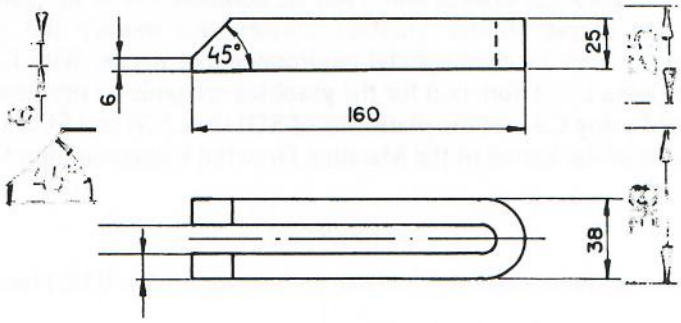


Figure 14. Graphics recognition by the GCL within MDUS: original image (in black) and detected dimension sets (in gray)

in detecting graphic objects in engineering drawings, as shown in Figures 12–14.

Figure 12(a) is part of a large noisy real life drawing we obtained from a very big European concern for test. In Figure 12(b) we successfully detect almost all bars, leaders, and text

(horizontal, vertical, and slant). In Figure 12(c) we successfully detect the two groups of hatching lines. Figure 13(a) is an ANSI [38] drawing with many solid and dashed circles. The result of line detection by MDUS is displayed in Figure 13(b) in solid lines with a single line width. All solid arcs and circles are correctly detected, while several arcs are false alarms. Three out of the four small dashed circles are correctly detected, though two of them are not entirely closed. The fourth small dashed circle at the bottom right is not detected because its top left dash is too long. Even the biggest dashed circle outside the biggest solid circle is correctly detected, though it is broken in two parts. The top part is longer than 3/4 circle and the bottom one consists of three dashes. All eight straight slanted dashed lines and six dash-dotted lines marking the hole centers are also correctly detected. Another detected dashed line is a false alarm, caused by joining the thick bar with a tail of a leader at the bottom right of the drawing. In Figure 14 we show the correct detection of dimension sets (including the radial dimension), displayed in a light gray color.

Other functions defined in the GCL may be called, and the graphics classes definition and implementation are also reusable. Users who wish to detect objects of new graphics classes can derive the new graphics classes from appropriate classes in the GCL. They may also define new features or modify the existing features of the graphics classes by overriding their member functions. By so doing, they may modify the details of the graphic recognition process within the defined framework in GCL.

SUMMARY

We have developed a Graphics Class Library (GCL) as a vertical reusable software component for graphics recognition. The library includes classes of graphic objects that appear in engineering drawings, as well as in other classes of line drawings. The purpose of establishing such a graphics library is to provide a framework for basic recognition algorithms of these graphics classes. The most important aspect of GCL is that it encompasses a generic graphics recognition algorithm. All graphics recognition processes are based on this generic algorithm. The code in the GCL is highly reusable and extendible. The GCL can be incorporated into other systems if they follow the same generic integrated graphics recognition framework. They can derive new graphics classes and override components of the generic recognition algorithm. They can derive similar graphics classes and modify the generic graphics recognition process to cater for their special requirements or needs. With the GCL, the user only needs to write a `main()` function for the graphics recognition program or system. The GCL is implemented using C++ on the platforms of SGI (Irix 5.3) and SUN (Solaris 2.5), and successfully operates as the kernel of the Machine Drawing Understanding System (MDUS).

REFERENCES

1. P. Freeman, 'A perspective on reusability', *Tutorial: Software Reusability*, IEEE Press, Washington, 1987, pp. 2–8.
2. *Khoros Manual*, University of New Mexico, 1991.
3. *Matlab: High-Performance Numeric Computation and Visulization Software: Reference Guide*, Math Works, Natick, MA, 1992.
4. *Programming with the Microsoft Foundation Class Library*, Microsoft Corporation, Redmond, WA, 1994.
5. J. W. Hooper and R. O. Chester, *Software Reuse: Guidelines and Methods*, Plenum Press, New York, 1991.
6. W. Liu and D. Dori, 'Automated CAD conversion with the machine drawing understanding system', *Proc. of 2nd IAPR Workshop on Document Analysis Systems*, Malvern, PA, USA, October 1996, pp. 241–259.
7. D. Dori, 'Object-process analysis: Maintaining the balance between system structure and behaviour', *Journal of Logic and Computation*, **5**(2), 227–249 (1995).

8. D. Dori and M. Goodman, 'From object-process analysis to object-process design', *Annals of Software Engineering*, **9**, 1–25 (1996).
9. W. Liu and D. Dori, 'Extending object-process diagrams for the implementation phase', *Proc. of the Third International Workshop on the Next Generation of Information Techniques and Systems*, Neve Ilan, Israel 1997, pp. 207–214.
10. J. Cohen, 'GTE software reuse for information management systems', *Proc. of the Reuse in Practice Workshop*, J. Baldo and C. Braun, Software Engineeering Institute, Pittsburgh, PA, 1989.
11. P. Coad and E. Yourdon, *Object-Oriented Analysis*, 2nd Ed., Yourdon Press, Prentice-Hall, Englewood Cliffs, NJ, 1991.
12. I. Jacobson, M. Christensson, P. Jonsson and G. G. Overgaard, *Object-Oriented Software Engineering*, Addison-Wesley, Reading, MA, 1992.
13. G. Booch, *Object-Oriented Analysis and Design with Applications*, 2nd Ed., Benjamin-Cummings, Redwood, CA, 1994.
14. P. T. Ward, 'The transformation schema: An extension of the data flow diagram to represent control and timing', *IEEE Trans. on Software Engineering*, **12**(2), 198–210 (1986).
15. R. Prieto-Diaz, 'Domain analysis: An introduction', *ACM Software Eng. Notes*, **15**(2), 47–54 (1990).
16. R. S. Conker, 'Dual plane variation of the Hough Transform for detecting non-concentric circles of different radii', *Computer Vision, Graphics and Image Processing*, **43**, 115–132 (1988).
17. D. Dori, 'Vector-based arc segmentation in the machine drawing understanding system environment', *IEEE Trans. on PAMI*, **17**(11), 1057–1068 (1995).
18. Y. Chen, N. A. Langrana and A. K. Das, 'Perfecting vectorized mechanical drawings', *Computer Vision and Image Understanding*, **63**(2), 273–286 (1996).
19. D. Pao, H. F. Li and R. Jayakumar, 'Graphic feature extraction for automatic conversion of engineering line drawings', *Proc. 1st International Conference on Document Analysis and Recognition*, Saint-Malo, France, 1991, pp. 533–541.
20. L. Boatto *et al.*, 'An interpretation system for land register maps', *IEEE Computer*, **25**(7), 25–32 (1992).
21. P. Vaxiviere and K. Tombre, 'Celesstin: CAD conversion of mechanical drawings', *IEEE Computer*, **25**(7), 46–54 (1992).
22. S. H. Joseph and T. P. Pridmore, 'Knowledge-directed interpretation of mechanical engineering drawings', *IEEE Trans. on PAMI*, **14**(9), 928–940 (1992).
23. C. P. Lai and R. Kasturi, 'Detection of dashed lines in engineering drawings and maps', *Proc. 1st International Conference on Document Analysis and Recognition*, Saint-Malo, France, 1991, pp. 507–515.
24. J. Gao, L. Tang, W. Liu and Z. Tang, 'Segmentation and recognition of dimension texts in engineering drawings', *Proc. of 3rd International Conference on Document Analysis and Recognition*, Montreal, Canada, 1995, pp. 528–531.
25. H. Luo, G. Agam and I. Dinstein, 'Directional mathematical morphology approach for line thinning and extraction of character strings from maps and line drawings', *Proc. of 3rd International Conference on Document Analysis and Recognition*, Montreal, Canada, 1995, pp. 257–260.
26. D. Dori and I. Chai, 'Extraction of text boxes from engineering drawings', *Proc. SPIE/IS&T Symposium on Electronic Imaging Science and Technology, Conference on Character Recognition and Digitizer Technologies*, San Jose, CA, 1992, pp. 38–49.
27. C. P. Lai and R. Kasturi, 'Detection of dimension sets in engineering drawings', *IEEE Trans. on PAMI*, **16**(8), 848–855 (1994).
28. A. K. Das and N. A. Langrana, 'Recognition of dimension sets and integration with vectorized engineering drawings', *Proc. of 3nd International Conference on Document Analysis and Recognition*, Montreal, Canada, 1995, pp. 347–350.
29. *AutoCAD Customization Manual, Release 12*, AutoDesk, Switzerland, 1992.
30. B. Smith and J. Wellington (eds.), *Initial Graphics Exchange Specifications (IGES), Version 3.0*, U.S. Department of Commerce and National Bureau of Standards, Gaithersburge, MD, 1986.
31. W. Liu, D. Dori, L. Tang and Z. Tang, 'Object recognition in engineering drawings using planar indexing', *Proc. of the First International Workshop on Graphics Recognition*, Pennsylvania, State University, PA, USA, 1995, pp. 53–61.
32. ftp.technion.ac.il/pub/supported/ie/dori/MDUS/libGCLsgi.tar.gz and libGCLsun.tar.gz.
33. D. Dori, W. Liu and M. Peleg, 'How to win a dashed line detection contest', in R. Kasturi and K. Tombre (eds.), *Graphics Recognition – Methods and Application*, *Lecture Notes in Computer Science*, Springer-Verlag, vol. 1072, 1996, pp. 286–300.

34. D. Dori and W. Liu, 'Vector-based segmentation of text connected to graphics in engineering drawings', in P. Werner, P. Wang and A. Rosenfeld (eds.), *Advances in Structural and Syntactical Pattern Recognition, Lecture Notes in Computer Science*, Springer, vol. 1121, 1996, pp. 322–331.
35. W. Liu and D. Dori, 'Recognition of hatching lines in engineering drawings', *Proc. 13th Israeli Symposium of AI, CV, and NN*, Tel Aviv, Israel, 1997.
36. W. Liu and D. Dori, 'Incremental arc segmentation algorithm and its evaluation', *IEEE Trans. on PAMI*, **20**(4), 424–431 (1998).
37. W. Liu and D. Dori, 'A generic integrated line detection algorithm and its object-process specification', *Computer Vision and Image Understanding*, **70**(3), 420–437 (1998).
38. *Dimensioning and Tolerancing*, ANSI Y14.5M, 1982.

