2000

Concept-based retrieval of object-oriented class components for software reuse.

Yu. Shen
University of Windsor

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

CONCEPT-BASED RETRIEVAL OF OBJECT-ORIENTED CLASS COMPONENTS FOR SOFTWARE REUSE

By

Yu Shen

A Thesis
Submitted to the College of Graduate Studies and Research
through Computer Science
in Partial Fulfillment of the Requirements for
the Degree of the Master of Science at the
University of Windsor

Windsor, Ontario

Abstract

It has been recognized that software productivity and quality can be improved through software reuse. Software reuse at all levels concerns about identifying and locating the potential reusable components. Reusability is considered as one of the great promises of OO technology. OO programming (OOP) especially supports software reuse via sub-classing through the forms of class inheritance and inclusion polymorphism. Objects and classes are basic building blocks in objected-oriented software development.

Having identified software components which are potentially reusable and described in such a way that anyone wish to reuse them would be able to do so, the problem arises as to how to organize the total collection of all such components and related descriptions. Such a reuse library can be constructed by classifying the reusable components in many different ways. A good system classification not only provides the basis for cataloging the components, but also provides a means for finding a particular candidate held in the reuse library. This thesis describes a methodology of classifying, structuring and retrieving object-oriented classes based on a mathematical method for data analysis called Formal Concept Analysis (FCA). In this methodology, a class is classified and retrieved using the type information plus variable access behavior of methods (in short, composite type) available in the class, and a class library is then viewed as a many-valued context. This many-valued context is transformed into a one-valued context, incorporating behavior of methods into the context via conceptual scaling, and Ganter's concept-generation algorithm is applied to this context, which leads to a lattice structure of all concepts.

Class retrieval is based on the concept lattice constructed from the class library. The concept-based retrieval system (CBRS) is implemented as a distributed system using CORBA with Java/Swing. User-friendly GUIs (Graphic User Interface) are provided for input, output, browsing class library, building concept lattice, and server status control. Before the users can retrieve anything from a class library, they need to specify what they need. The system organizes the user's specific requirements into a query. Choosing one or more composite types (the desired behavior) available from the list will reduce the total numbers of available composite types. Each chosen composite type refines the query. Repeating the above steps eventually narrows search target to one or more concepts whose intents have all necessary composite types requested by the

user. The users can modify the query's ingredients. Once the user satisfies with query, a vector that holds all the information the user needs will be passed to the retrieve engine. The result is passed back immediately to the client when the retrieve engine finishes processing the user query. The query formulation is incredibly easy. The query constructed by the user in our CBRS has clear semantics, thus the result is comprehensible to the user. The methodology combines of the exact semantics of formal methods and interactive navigation possibilities of informal methods.

An incremental selection from the list of available composite types guarded by the retrieval system allows exact match and completely eliminates the data mismatch and time-consuming deduction phase in a traditional non-concept-based retrieval schemes, and thus improves retrieval reliability. The lattice structure not only improves the retrieval efficiency by searching only relevant concept space but also allows more precise information to be retrieved from the library because of the clear semantics of the user query. In other words, the concept lattice allows retrieve engine to process queries more efficiently and provides optimal feedback to the user.

The concept-based retrieval scheme proposed in this thesis reduces retrieval complexity and cost yet still maintain high precision and recall comparing with other approaches. The key is to impose a structural order of all the classes in a class library. This can only be achieved through a many-valued context to a single-valued context and then to a concept lattice representation of the class library. Mixing type and behavior of methods of a class actually makes our concept-based methodology a combination of type-based and behavior-based retrieval approaches. Furthermore, our methodology is flexible enough to allows us to incorporate more class attributes in terms of formal indices and informal descriptions into the context, so the potential is very promising.

The facility of browsing a class library is also provided in our CBRS system. The purpose of browsing a class library is two folds: (1) the users are not familiar with the retrieval system and have no concrete knowledge what they are looking for; (2) the users want to go through browsing a class library to achieve the purpose of retrieval of partially-satisfied classes because there is not always an exact match between the user's requirements and available classes in the class library. In both cases, the user wants to look around for a class that is close enough to be adapted to

his/her needs. The hierarchy generated from the concept lattice of a class library can be used for browsing and understanding the relationships between the classes.

Navigating directly on a concept lattice hierarchy of a class library actually plays the roles of browsing and searching the library. The navigating is achieved through an applet called **ConceptLattice** that is implemented in Java/Swing **JTree** data structure. The Swing **JTree** is a perfect choice to display hierarchical data. The advantage of using concept lattice is that we can incorporate more attributes to a class like sample behavior or a very good class description for each node (concept) that will also present to the user, which greatly enhances the usefulness of the hierarchy for browsing purpose.

It is the first time that the methodology of concept-based retrieval of object-oriented classes from a class library and its full integrated implementation are provided as a distributed client/server CORBA applications.

Dedication

This thesis is dedicated to my wife, Daihong and my son, Tony, for their love, understanding and support.

Acknowledgments

First of all, I am greatly indebted to Dr. Park for his excellent instruction and advice on the thesis-related research work. Many valuable discussions with him and his excellent suggestions greatly stimulated my interest in component-based technology, which not only significantly contribute the quality of this thesis, but also benefit my career-pursuing endeavor. Dr. Park's excellent teaching and advice on my courses certainly impressed me very much and shortened the process of being a Master qualifying candidate, respectively.

Many thanks go to Dr. Chen, my internal reader, for providing innumerable suggestions and comments for improvement.

Special thanks to my external reader, Dr. Kao who showed immense interest in my work, provided many suggestions and comments which certainly improve the quality of this thesis.

Thanks also due to the support team of Visibroker for Java at Inprise Corporation, whose prompt and useful feedback helped me set up the great ORB for running quickly.

Finally, I must thank my family members, Daihong and Tony, for the sacrifices that were made while I was working on my thesis.

TABLE OF CONTENTS

List of Figures

List of Tables

Chapter 1 Introduction

1.1 What is Software Reuse?

Software reuse is the process of implementing or updating software systems using existing software assets, or more broadly, software reuse is the application of reusable (designed for reuse) software assets to more than one software system. It is a software development strategy that attempts to reduce development costs and to improve software quality by incorporating previously proven work products into a new software product.

Software reuse may occur within a software system, across similar software systems, or in widely different software systems. Reuse refers to software components as well as other engineering and development products. A reusable component can be a few line of code, or a broader and high level entity such as an object, module, or encapsulated collection of functionality, services, software specifications, designs, tests cases, data, prototypes, plans, documentation, frameworks, and even templates. Software reuse has two sides: (1) systematic development of reusable components (for example, a reuse-driven software development process). (2) systematic reuse of these components as building blocks to create new systems (for example, the beans of JavaBean can be building blocks to create more complex component). Technically there are many types of reuse at different productivity levels such as code reuse, inheritance reuse, template reuse, component reuse, framework reuse, artifact reuse, pattern reuse and domain component reuse in the second category.

The success of the software reuse process depends on many factors. But there are three important ones from point view of technique according to Berzins and Luqi (1991):

(1) Existence of a software repository containing many instances of each type of software components, and availability of incremental procedure for updating the repository.
(2) Availability of efficient algorithms for describing and indexing the components, and finding appropriate components from a possibly large software base.
(3) Reusability of existing components.

1.2 Software Reuse Techniques

Software development with reuse includes both business (organizational activities including management of reusing a program, market place analysis, financing and marketing forecast, and training) and engineering/technical (technical activities including technologies or tools that support reuse, for example, CASE (computer-aid software engineering) tools and common interfaces like CORBA (Common Object Request Broker Architecture), DCOM (Distributed Component Object Model), and COTS (Commercial Of The Shelf), software development with reuse processes or technical procedures like domain modeling, product-line approach, common architecture, quality control, and best development practice) activities. There are many factors that influence successful reuse, among which management of an organization plays very important role. However, here we will restrict ourselves to technical aspects of software development with reuse. The reuse techniques encompass a wide range of activities during system development. Reuse can be roughly classified into two categories from point of view of reuse approach:

(1) Model-based reuse: Application is constructed not by reusing objects directly but using standardized basic model (frameworks and concepts). This is the highest level reuse. Application developers can take advantage of the common (abstract) models and derive from them the specific models or parts appropriate for their environment. The examples of the technique are use of domain model, reference model, pattern and so on. If we enlarge and apply this reuse technique not only to models but also to the way of analyzing and designing, methodology and CASE tools could be parts of model-based reuse. It is the advantage of this approach that the abstract level of the common model is so high that those models are easy to understand and to be applied to a wide range of

applications (i.e., those model are designed for reuse, it is a part of reuse-driven software development effort). On the contrary, this approach requires more workload to modify the common models and apply them to the real environment.

(2) Component-based reuse: This technique is to construct application by using offered parts without any change of them or with slightly modification. Developers search from reuse library for suitable parts to compose the application. The examples of the technique are use of class libraries or use of JavaBean bean repositories. By this technique, we use parts directly , so we have advantage of less work in operation. On the other hand, the ease of direct use of parts may depends on platform and its language. Object-oriented (OO) class libraries are generally designed for the component-based reuse or parts-based reuse. When inheritance mechanism is considered, however, they may also be regarded as a kind of model-based reuse.

1.3 Retrieval Approaches

It has been recognized that software productivity and quality can be improved through software reuse. Software reuse at all levels concerns about identifying and locating the potential reusable components. Reusability is considered as one of the great promises of OO technology. OO programming (OOP) especially supports software reuse via sub-classing through the forms of class inheritance and inclusion polymorphism. Objects and classes are basic building blocks in objected-oriented software development.

Having identified software components which are potentially reusable and described in such a way that anyone wish to reuse them would be able to do so, the problem arises as to how to organize the total collection of all such components and related descriptions. Such a reuse library can be constructed by classifying the reusable components in many different ways. A good system classification not only provides the basis for cataloging the components, but also provides a means for finding a particular candidate held in the reuse library. Large collection of software present similar problems of classification to those of information retrieval (Salton and McGill 1983). In the past decade, most efforts from

research community have concentrated on software classification, with emphasis upon domain-specific aspects such as the signature of operations/functions, and determination of static metrics that can be used when searching and retrieving. During this period many researchers applied library classification methods to software component libraries.

There are several ways to retrieve reusable components from reuse library, such as type-based, semantics-based, composition-based and execution-based retrievals. The retrieval approaches proposed in the literature basically fall into three categories, based on the techniques used to index the library components:

(1) External Indices (classification-based schemes): keyword, faceted, knowledge-based and feature-oriented classification schemes (Hennigger 1994; Maarek, Berry and Kaiser 1991; Prieto-Diaz 1987 and 1991; Yau and Tsai 1987; Borstler 1995) usually use controlled vocabularies, properties, ontological features to extract relevant components.

(2) Internal Static Indices (structural schemes): those schemes use structural characteristics such as signature matching, specification matching to seek relevant components (Park and An 1998). The signature matching scheme allows both exact and relaxed match, and can be operated in function matching (where signature = type) and module matching (where signature = a multiset of user-defined types and a multiset of function signatures) models.

(3) Internal Dynamic Indices (behavioral schemes): behavior-based schemes compare desired input and output with those of executing components to select relevant components (Atkinson and Duke 1995; Niu and Park 1999). Another type of behavior-based retrieval is to compute similarities between user-input query and semantic networks both of which describe behaviors of relevant components (Chou, Chen and Chung 1996).

Application of formal specification methods to software libraries has also been investigated (Katz et al 1987; Perry 1987; Rollins and Wing 1991). In these approaches, each component is indexed with a formal specification which captures its

relevant behavior. Any desired relation between two components is expressed by a logical formula composed from indices. An automated theorem prover is usually used to validate the formula. If and only if the prover succeeds the relation is considered to be established. These specification-based retrieval approaches (Fischer et al 1998; Zaremski et al 1995; Pinix et al 1995; Mili et al 1997; Schumann and Fischer 1997) allow arbitrary specifications as keys and retrieval all components from a library whose indexes satisfy a given match relation with respect to the key. These strategies are all based on a simple idea: reducing the search scope by matching the arbitrary specifications to retrieve candidate classes. Unfortunately, their results are often a partial match. Fischer et al called these strategies deduction-based retrieval techniques (Fischer, Schumann and Snelting 1998), which are very time-consuming retrieval approaches.

Much effort has been made in the area of finding appropriate classes for reuse. A helper class for a class library allows user to specify key words to search the library for a match (McManis 1996). Damiani et al reported a descriptor-based retrieval approach (Damiani, Fugini, and Fusaschi, 1997). Liao and Wang attempted to improve searching by reorganizing the OO library with facet classification scheme and thesaurus (Liao and Wang 1993). Nelson and Poulis developed a CSRS system which employs the OO DBMS to query the specific classes (Nelson and Poulis 1995). These strategies are called deduction-based retrieval techniques and very time-consuming retrieval approaches (Fischer, Schumann, and Snelting, 1998). Atkinson and Duke proposed an abstract methodology for behavioral-based retrieval of class library (Atkinson and Duke 1995). In their methodology, class retrieval focuses on the specification of the theoretical frameworks for behavioral retrieval, without any optimized algorithm for practical implementation. Niu and Park has developed an execution-based retrieval of class library in which the system organizes the user-input test data into a test program and executes each test message on each class in a class library to find the relevant classes based on the number of matched and unmatched methods (Niu and Park 1999).

This method actually executes all classes in the class library, therefore has some potential problems.

Formal Concept Analysis (FCA) is a mathematical method for data analysis based on the lattice theory of Garret Birkhoff (Birkhoff 1993). It takes unstructured data and transforms into well structured units which are formal abstraction of concepts of human thoughts. The clear representation of data allows investigation and meaningful, comprehensible interpretation (Ganter and Wille 1999). Lindig was the first who applied FCA to construct a component library which a concept-based method of retrieving Unix system calls indexed by keywords was based on (Lindig 1995). An overview of the concept analysis as a new framework and its application in software engineering areas such as analysis of configuration spaces (Snelting 1996) and modularization of legacy code (Lindig and Snelting 1997; Siff and Reps 1997) has been reported (Snelting 1998). More recently, Fischer has applied FCA to construct a specification-indexed software component library in which the user retrieves the desired components by navigating the concept lattices directly (Fischer, Schumann and Snelting 1998). Park described a method of retrieving software components using samples based on FCA (Park 1999). A more detailed overview of FCA application in software engineering area can be found in chapter 2.

1.4 Overview of the Thesis and Motivation of Concept-Based Retrieval

As mentioned above Niu (Niu 1999) has extended the methodology proposed by Atkinson and Duke (Atkinson and Duke 1995) originally for retrieval of function component to object-oriented class library. Niu's methodology has improved the retrieval precision because of execution of each test message on each class of the class library. His retrieval schemes heavily rely upon executing each method of each class in the class library on test data provided by the user to capture the class behaviors and compare these captured behaviors with desired ones for match. It is first time that this methodology of

class retrieval by executing a class and its full implementation has been established. However, there are some problems not dealt with in his methodology:

(1) **Cost and Efficiency** In order to retrieval desired candidate classes the system is forced to execute each class in the class library, whose behaviors are closest to the user-specified behaviors. For a small library containing up to a few hundred class files the total time spent on executing each class maybe acceptable. However, execution on each class is too expensive for a huge-volume class library which is ordinary in most of software development organization although Niu has attempted to type-check on both input and output to reduce number of classes the system has to loop through. Based the above observation and lack of any data structures in the class library, the efficiency of the retrieval system can't be high.

(2) **Arguments Order is Required** When user constructs a test message, he or she must specify the order of arguments within the test message same as the order of a method of a class for a successful match. This requirement is not practical because the user may not know the order of a method's arguments or the relative order is not important.

(3) **Limited Input Types** The user is responsible for constructing test message, composed of a set of actual input and output values including associated type information, which is to be sent to a class for execution. The types accepted in Niu's retrieval system are only primitive data types defined by Java language, the user-defined data types are not allowed. This limitation certainly will narrows its usability.

(4) **Uncaptureable Behavior** A class' behaviors are mainly defined by its methods. Not all the behaviors of a class can be captured by Niu's methodology. For example, suppose a class has two methods: one method accepts user's input, and only updates the class's hidden variables and returns nothing (similar to a non-empty arguments class constructor), another method returns value. This kind of behavior of the class can't be captured by the retrieval system due to lack of inner relation between one test message and another test message within one test program constructed by the user, therefore there is no way to retrieve this class.

In this thesis, a new methodology is proposed which is based on the FCA. This is a complete different approach than Niu's methodology. The main points of our methodology are summarized as following:

- The proposed methodology is based on formal concept analysis. This all-new data analysis tool will be used to organize our class library. Previous work has shown that a structured library will greatly improves retrieval efficiency, particularly execution-based retrieval scheme. An organized class library using FCA mainly consists of a partially ordered set of all the concepts of a given context that represents the class library. Ordering classes in a class library will make retrieval more efficiently.
- In this methodology, each class in a class library is classified using the type information plus variable access patterns of all the available variables/methods in the class, and then the class library is viewed as a many-valued context. This many-valued context is transformed into a one-valued context, incorporating behavior of methods into the context via conceptual scaling, and Ganter's concept-generation algorithm (Ganter 1986) is applied to this context, which leads to a lattice structure of all concepts. Mixing type and behavior of methods of a class actually makes this methodology a combination of type-based and behavior-based retrievals. Furthermore, this methodology is flexible enough to allow us to incorporate more class attributes like sample behavior into the context, so the potential is very promising.
- Class retrieval is based on the concept lattice using the type information plus variable access behavior of available methods of a class in a class library. The methodology provides a way to organize the user's specification into a query. The users can modify the query's terms. Once the user satisfies with query, a vector that holds all the information the user needs will be passed to the retrieve engine. The result is passed back immediately to the client when the retrieve engine finishes processing the user query. The query constructed by the user in this methodology has clear semantics, thus the result is comprehensible to the user.
- This methodology also provides a way of an incremental selection from a list of available composite types by user, which allows exact match and completely eliminates the data mismatch and time-consuming deduction phase in a traditional non-concept-

based retrieval schemes, and thus improves retrieval reliability. The lattice structure not only improves the retrieval efficiency by searching only relevant concept space but also allows more precise information to be retrieved from the library because of the clear semantics of the user query. In other words, the concept lattice allows retrieve engine to process queries more efficiently and provides optimal feedback to the user.

- The facility of browsing a class library is also provided in our methodology. The purpose of browsing a class library is two folds: (1) the users are not familiar with the retrieval system and have no concrete knowledge what they are looking for; (2) the users want to go through browsing a class library to achieve the purpose of retrieval of partially-satisfied classes because there is not always an exact match between the user's requirements and available classes in the class library. In the late case, the user wants to look around for a class that is close enough to be adapted to his/her needs. The hierarchy generated from the concept lattice of a class library can be used for browsing and understanding the relationships between the classes.
- Navigating directly on a concept lattice hierarchy of a class library actually plays the roles of browsing and searching the library. The navigating is achieved through an applet that can be implemented in Java/Swing **JTree** data structure. The Swing **JTree** is a perfect choice to display hierarchical data. The advantage of using concept lattice is that we can incorporate more attributes to a class such as a sample behavior, a very good class description for each node (concept) that will also present to the user, which greatly enhances the usefulness of the hierarchy for browsing purpose.
- A distributed CORBA client/server applications implemented using Java and Java/Swing not only make its easy to put the whole system on the Internet and enterprise Intranet, which enable the user perform remote method calls, but also provides professional graphic user interfaces.

The FCA provides a method for turning a traditional description-based class library into a well defined class hierarchies. Each chain of concepts from the top of the lattice to the bottom contains classes with minimal difference between neighbors. This produces maximally deep class hierarchies which greatly facilitates retrieving desired candidate

classes. Many advantages (see part 5.3 of chapter 5 for a detailed summery) of this methodology plus the unique way of extracting type and access behavior of methods from a class allows us to use this methodology to solve the first three problems very well and partially solve the last problem mentioned early in this part. The eventual goal of our research is to specify a class-based software reuse repository by integrating a number of retrieval approaches including typed-based, execution-based and attribute-based schemes with other repository operation. This work is a part of above effort.

1.5 Organization of the Thesis

This thesis is organized into five chapters. The following is a brief description of each chapter concerns about.

Chapter 1 provides a brief introduction of software reuse, reuse models, retrieval techniques of various components, and overview of this thesis.

Chapter 2 presents an overview of formal concept analysis and its applications in software engineering area. It also provides a short introduction of mathematical foundation of the formal concept analysis with basic definitions, notations and theorems to thoroughly understand our FCA-based methodology.

Chapter 3 discusses general ideas of concept-based retrieval. A methodology of construction, organization, building concept lattice from a one-valued context scaled from a many-valued one of a class library is presented. Retrieving classes based on concept lattice through formulation of user queries from a model class library is demonstrated. Two small scale class libraries are also presented to illustrate the procedure of organization of contexts, construction of concept lattices and so forth.

Chapter 4 describes a prototype of concept-based retrieval system (CBRS) implemented using CORBA and Java/Swing. A distributed CORBA client/server application is discussed. All the components of the distributed CBRS system are systematically described in detail. The screen shots of important GUIs of both client and server sides are provided. An incremental selection mechanism for formulating user query and retrieval approaches are discussed. A class browsing tool based on navigation of concept lattice is also introduced in this chapter.

Chapter 5 compares our methodology proposed in this thesis with former researches and other retrieval tools. It also covers problems left in this thesis and possible future works in this new area.

Chapter 2 Formal Concept Analysis and Its Applications

2.1 Introduction

Formal concept analysis provides a conceptual tool for the analysis of data, and has already had many successful applications (see the part 2.2 below for a review). One of the main objectives of this method is to visualize the data in form of concept lattices and thereby to make them more transparent and more easily discussible and criticizable. Another important tool provided in connection with formal concept analysis is the method of "interactive attribute exploration", which allows knowledge acquisition from (or by) an expert by putting very precise questions to him/her, which either have to be confirmed or to be refuted by a counterexample.

Formal concept analysis transforms any binary relation constructed from a set of objects and a set of attributes into a complete lattice. This concept lattice can be studied by algebraic means and offers remarkable insight into properties and structure of the original relation. As relations between "objects" and "attributes" occur all the time in software technology, formal concept analysis is becoming an attractive foundation for a new program analysis and design tools.

2.1.1 What is Formal Concept Analysis (FCA)?

FCA was founded by Wille (Wille 1982; Davey and Priestley 1990). It is basically a data analyzing tools that can be used to calculate a concept lattice from a binary relation of objects and attributes, called a formal context. FCA is a set-theoretical model for

concepts that reflects the philosophical understanding of a concept as a unit of thought made up of two parts: the extension and the intention. The extension of a concept is the collection of all entities (objects) belonging to the concept and the intention is all the properties (attributes) common to all entities of the concept.

The concept lattices is a structure with strong mathematical properties which reveals the hidden structural and hierarchical properties of original relation. It can be computed automatically from any given binary relation.

2.1.2 Application Domains of Formal Concept Analysis

FCA provides a way to identify groupings of objects that have common attributes. The mathematical foundation was laid by Birkhoff in 1940 (Birkhoff 1940). Later, Wille and Ganter elaborated Birkhoff's result and transformed it into a data analysis method (Wille and Ganter 1999; and Wille 1982). Since then FCA has found a variety of applications.

The FCA is widely used in numerous application domains, e.g. in psychology where repertory grids were analyzed using FCA; in information science where FCA was used to construct a retrieval system; in software reengineering where FCA was used to locate clusters of subroutines in 20 years old FORTRAN code; in software engineering where FCA was used to help class hierarchy design; the FCA was also applied to reengineer, represent and investigate knowledge bases created by ripple down rules in an iterative manner (Richards and Compton 97); the FCA was also applied to the software reuse where a concept lattice was constructed from a reuse library to help retrieve reusable components; and in many other areas such as software engineering, musicology, politics, linguistics, electronics, civil engineering, ecology, biology, and psychology. We will go over some application areas such as software engineering, reengineering, class design and software reuse in details in next part.

2.2 Application of FCA in Software Engineering

Software reengineering and reuse are concerned with maximizing software usage for any given development effort. When a large and important family of software products gets out of control, a major effort to restructure it is appropriate. The first step must be reduce the size of the program family. One must examine the various versions to determine why and how they differ (Parnas 1994). However, at that time there were very few even no good method available for reengineering program families, let alone tool support for restructuring. At the same time, Krone and Snelting reported a first step toward a theory and tools for configuration restructuring, and they have shown how configuration structure can be inferred from existing source code and how interference between configurations can be detected based on FCA (Krone and Snelting 1994). Since then FCA is quickly emerging as an important tool for software reengineering. In 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tool and Engineering, Snelting (Snelting 1998) presented a brief overview of the FCA as a new framework and its application in software engineering areas including software component retrieval (Lindig 1995), analysis of configuration spaces and modularization of legacy code (Lindig and Snelting 1997; Siff and Reps 1997). In this part we will review the applications of FCA in the software engineering (including reengineering), and then software design.

2.2.1 Software Engineering & Reengineering

The practical application of using FCA has resulted in many developments of concerning the visualization of the lattice using computer generated diagrams (Wille 1984), its simplification through decomposition or pruning heuristics (Ganter and Wille 1999; Godin and Mili 1993; Funk, Lewien, and Snelting 1995), its use in an interactive knowledge acquisition process and the generation of rules from the lattice (Wille 1992)

which can be used for knowledge discovery in databases (Godin and Missaoui 1994). Recently applications of FCA to several software engineering areas have been reported ever increasingly (Lindig 1995; Snelting 1996; Snelting 1998; Snelting and Tip 1998; Lindig and Snelting 1997; Siff and Reps 1997; Krone and Snelting 1994; Godin, Missaoui and April 1993; Godin, Mineau and Missaoui 1995; Fischer 1998; Park 1999; Shen and Park 1999).

As early as 1986, Godin et al had studied lattice model as browsable data spaces in data analysis (Godin, Sanders and Gecsei 1986). A structure very similar to the concept lattice had also been proposed for retrieval of classes in a large repository based on class features in 1992 (Oosthuizen, Bekker and Avenant 1992). In 1995, Lindig reported an application of FCA to retrieve Unix system calls from a small library in which the Unix system calls were individually indexed by carefully selected key words. A concept lattice was constructed based on a table (i.e., context) consisting of the Unix system calls as a set of objects and the key words as a set of corresponding attributes; He demonstrated clearly, for the first time, the concept lattice allowed a fast, exact and incremental retrieval of the Unix system calls; Queries could be easily constructed and have clear semantics, thus the obtained results were comprehensible to the user (Lindig 1995).

At about same time, Godin et al also applied FCA to a database dictionary which was considered as a standard reference for building database schemas and ensuring uniformity and standardization within a large company's applications. When a new application was defined, the analyst started first by determining if existing data element satisfied the needs of the application through retrieval of the data element. The retrieval process was aided by browsing the concept hierarchy generated by using FCA (Godin, Mineau and Missaoui 1995). In another paper, Godin et al also performed an experiments to compare information retrieval using a Galois lattice structure with two more conventional retrieval methods -- navigating in a manually built hierarchical classification and Boolean querying with index terms. Their research results found no significant performance difference

between the former and the laters. This finding suggested that retrieval using concept lattice structure could be an attractive alternative for information retrieval (Godin, Missaoui and April 1993).

Eklund's research group at Griffith University have conducted a series of research using FCA to aid information retrieval since 1996. Cole and Eklund reported that they used SNOMED (Systematized Nomenclature of Medicine) to index a set of 9,000 patient discharge summaries. A concept lattice is synthesized whose structure reflects both specialization/generation information present in SNOMED and combinations in which the SNOMED concepts appear in the documents. The concept lattice is synthesized using format concept analysis, where documents are considered objects and the medical concepts contained in SNOMED are considered the attributes (Cole and Eklund 1996a and b). More recently the same authors also applied FCA to analyze an email collection. They described the use of a suite of tools designed to allow an investigation of data retrieved from email texts. The data is retrieved from the emails with aid of a hierarchy of classifiers that extract useful terms and encode known implications. Further implications, both complete and partial, are then investigated by means of a nested line diagram (Cole and Eklund 1999).

Lindig and Snelting even applied FCA to modularize legacy code. By analyzing the relation between procedures and global variables, a concept lattice is constructed. Module candidates were identified and arranged in the concept lattice. They then explained how hierarchical clustering of local modules or procedures showed up as sub-/superconcept relation in the lattice, and how the lattice could be used to assess cohesion and coupling between module candidates. Specific infima (so-called interferences) corresponded to violations of modular structure, and proposals for interference resolution could be automatically generated. Furthermore, certain algebraic decompositions of the lattice could lead to automatic generation of modularization proposals. They also applied this

approach to several examples written in Modula-2, FORTRAN, and COBOL; among them a >100 kloc aerodynamics program was analyzed (Lindig and Snelting 1997).

In 1996, Snelting applied FCA to the problem of reengineering configuration. FCA restructured a taxonomy of concepts from a relation between objects and attributes. He used FCA to infer configuration structures from existing source code. They used a tool called NORA/RECS which accepted source code, where configuration-specific code pieces were controlled by preprocessor. The computed concept lattice was proved to provide remarkable insight into the structure and properties of possible configurations. The concept lattice not only displayed fine-grained dependencies between configurations, but also visualized the overall quality of configuration structures according to software engineering principles. Interferences between configurations could be analyzed in order to restructure or simplify configurations. Interferences showing up in the concept lattice indicated high coupling and low cohesion between configuration concepts. Sources files could be then simplified according to the lattice structure (Snelting 1996).

More recently in 1998, Snelting presented a short overview of the FCA's underlying theory, as well as its applications for software component retrieval, analysis of configuration spaces, and modularization of legacy code and other work done in his group. He concluded that the concept lattice can be studied by algebraic means and offers remarkable insight into properties and structure of the original relation. As relations between ``objects" and ``attributes" occur all the time in software technology, concept analysis is an attractive foundation for a new class of program analysis tools (Snelting 1998).

It is appropriate to mention that Lindig even applied FCA to another software reengineering area -- analysis of software variants caused by diversity of computer platforms existed in today's reality. His research results shown that the variants generated from a source have a rich structure which can be effectively analyzed by using FCA, and

variant description may contained redundant expression which can be removed (Lindig 1998).

Another paper recently published by Snelting and Tip reported applications of FCA to find design problems in a C++ class hierarchy by analyzing the usage of the hierarchy by a set of applications. The concept lattice is constructed in which relationships between variables and class members are made explicit, and where information that members and variables have in common is "factored out". They have shown that the FCA-based technique be capable of finding design anomalies such as class members that are redundant or that can be moved into a derived class. In addition, situations where it is appropriate to split a class can be detected. They even suggested the FCA-based techniques can be incorporated into interactive tools for maintaining and restructuring class hierarchies (Snelting and Tip 1998).

Fischer has applied FCA to specification-based retrieval system in the software component libraries. He has shown that match relations can be used to build appropriate index which then will be used to index the library and the FCA can be used to turn this index into a navigation structure. His experiments showed that its is feasible to calculate an approximation of the index which is accurate enough for browsing purposes, using current theorem provers and hardware. The concept lattice reveals the implicit structure of a library as it follows from the index. It can even indicate situations where a finer index is required and thus help understand and reengineer component library. Due to its dual nature, the concept lattice allows two complementary navigation styles which are based on either on attributes or on objects. Due to the lattice nature, both navigation styles automatically have the single-focus property and refrain the user from dead ends (Fischer 1998).

Research of Park's group the research is recently focused on component reuse through retrieval from a class library aided by FCA tool. Park has presented a method of indexing,

storing and retrieving potentially reusable class components. This method is based on representing classes using type information, constructing concept lattice structure for the library based on accessibility relation between classes and types using FCA, and retrieving reusable classes using types of methods of the desired class by navigating the concept lattice. The concept lattice structure certainly improves the search time comparing with searching unorganized library. The system allows user to incrementally select from a menu of available types guided by the retrieval system (Park 1999).

In another paper by Shen and Park, they proposed a methodology of classifying, structuring and retrieving object classes based on a mathematical method for data analysis i.e., FCA. In this methodology, a class is classified and retrieved using the type information and variable access behavior of methods in the class and a class library is then viewed as a many-valued context. This many-valued context is transformed into a one-valued context via conceptual scaling and FCA is applied to this context, which leads to a lattice structure of all concepts. Class retrieval is based on the concept lattice. The lattice structure improves the retrieval efficiency by searching only relevant concept space. The user can also select incrementally from a list of available type information plus variable access pattern of methods dynamically updated by the retrieval system. More other attributes of classes can be easily incorporated in the many-valued context for the class library, it allows more precise information to be retrieved from the class library (Shen and Park 1999). The methodology described in this paper forms the foundation of this thesis.

2.2.2 Software Design

Building and maintaining class hierarchy has been recognized as one of the most important and difficult activities on object-oriented software design (Booch 1994). Transforming an imprecise, natural-language description of a client's requirements into a well defined class hierarchy model is difficult and demanding task. The problem of

building an initial class hierarchy from a set of class specification or reengineering an existing one following class updates, has received increased attention in the OO literature in recent years. Class hierarchies start taking shape at the analysis level, where classes that share application-significant data and application-meaningful external behavior are grouped under more general classes. Rubin and Goldberg has proposed an object behavior analysis method which involves a sub-step where identifying the objects, their attributes and services, the class hierarchy is reorganized in order to represent relevant abstractions based on common behaviors and attributes (Rubin and Goldberg 1992). At the design level, such hierarchies are augmented and possibly reorganized in order to take into account the solution objects along with the application domain objects (Monarchi and Puhr 1992). Several guidelines have been proposed for the design of class hierarchies. Among these, two characteristics emerge from the literature that are particularly revenant the class design using FCA: (1) Minimizing redundancy -- There is large consensus that keeping each thing in one place in the class hierarchy is a good software engineering practice (Johnson and Foote 1988; Korson and Mcgregor 1992; Lieberherr, Bergstein and Silva-Lepe 1991). Redundancy may also means that the right abstractions have not been identified based on commonalties within the library. (2) Subclasses as specialization -- Inheritance is sometimes used only for code reuse purposes, which, as observed by Cox (Cox 1990), produces libraries that are difficult to understand and reuse. Many people advocate that inheritance hierarchy be consistent as possible with specialization to achieve better understandability and reusability (Johnson and Foote 1988; Coleman et al 1994).

There are quite few literature in the area of application of FCA to the class design. This is a quite new and uncharted territory in software engineering. Back to the old days, work in the context of the Demeter System has addressed the automatic discovery of class hierarchies from example objects (Lieberherr, Bergstein and Silva-Lepe 1991). Their spirit is similar to using FCA to generate class hierarchy. They use a class dictionary graph to represent the design. Their algorithm uses a metric which underlies an optimization process. They propose a two-step learning algorithm where the first step

does basic learning by generating a potentially non-optimal class dictionary graph. The edges of the graph represent the inheritance (alternation edges) and part-of (construction edges) relationships between classes. The second step optimizes the graph by trying to minimize a weighted function of the edge where the weight of the construction edge is at least twice the weight of the alternation edges. This function can be considered as a complexity metric for the class hierarchy. They have shown that the exact optimization is **NP hard** and proposed an approximate algorithm that brings down the complexity to polynomial. They also present an incremental algorithm that produces the optimal hierarchy when it is a tree (Bergstein and Lieberherr 1991). From today's point of view, the same goals can be easily achieved with FCA with an important advantage: the resulting class structure is not dependent on any algorithm that produced it (either batch or incremental algorithms will give the same structure of concept lattice!!).

Cook was more interested in critiquing and reorganizing existing class hierarchies to bring class (implementation) hierarchies as close as possible, language permitting, to type (interface) hierarchies (Cook 1992). In one experiment, he automatically extracted interface specifications of the Smaltalk-80 Collection class library to build the corresponding interface hierarchy, and examined the produced hierarchy to detect problems with actual library and proposed some improvements which are feasible within the context of Smalltalk's single inheritance.

Based on an empirical study, Dvorak showed how class hierarchies tend to exhibit conceptual entropy which is manifested by increasing conceptual inconsistency as we travel down the hierarchy (Dvorak 1994). As a solution, he proposed a method and an algorithm for building class hierarchies emphasizing conceptual simplicity and consistency. The algorithm relies on manually-generated formal specifications of the conceptual attribute for each class.

Notice that the focus of these algorithms is how to put the initially provided classes into a hierarchy. They do not address the restructuring of the basic objects and classes. There might be better alternative designs of these initial classes based on splitting or combining classes. Work on such behavior preserving transformation on class hierarchies have been reported (Opdyke and Johnson 1989 & 1993). The behavior-preserving transformation was called refactorings in their work. The goal of refactoring is to improve design and enable reuse by "factoring out" common abstraction. This involves steps such as the creation of new super classes, moving around methods and classes in a hierarchy, and a number of similar steps.

A complimentary technique to the one of Opdyke and Johnson, based on FCA, has been reported by Snelting's group. Again Snelting and his research group have been pioneered in reengineering C++ class hierarchies using FCA (Snelting and Tip 1998). In their paper, they presented a method for finding design problems in a class hierarchy by analyzing the usage of the hierarchy by a set of application. The concept lattice constructed using FCA reveals explicit relationships between variables and class members, where information that members and variables have in common is "factored out". Their results shown that the technique be capable of finding design anomalies such as class members that are redundant or that can be moved into a derived class. They also showed how a restructured class hierarchy can be generated from the concept lattice, and how the concept lattice can server as a formal basis for interactive tools for redesigning and restructuring class hierarchies.

Based on FCA, Davies presented an automated technique for constructing class hierarchies as concept lattices (Davies 1997). He even suggested a software development method using FCA in following steps: (1) Split systems into subsystem ... (2) For subsystem at class hierarchy implementation level: (a) Identify classes of subsystem. (b) Identify properties of these classes. (c) Create property cross table. (d) Generate class hierarchy from property cross table. (3) Refine by repeating steps 1 and 2. (4) Implement

public interfaces of the classes. At all times the concept lattice provides a visual representation of the system and all subsystems. The higher level concepts which become the abstract class and higher level classes of the implementation should be meaningful within the context of the subsystem and perhaps within the framework of the project as a whole. He also summarized the benefits of this technique which include automatic elimination of no-essential complexity, creation of deep class inheritance structures and visual representation of all classes within a project as one relational, mathematical structure.

Finally, Godin et al recently presented concept lattice and related structures as framework for dealing with design and maintenance of Smalltalk class hierarchies (Godin, Mili, et al 1998). They designed incremental algorithms that update existing concept lattices as the result of adding, removing, or modifying class specification due to the nature of an inherently iterative and incremental process in class design. They developed a prototype tool that incorporates this and other algorithms as part of the IGLOO project, which is a large object-oriented software engineering joint research project involving academic and industrial partners. Their tool can generate either the concept lattice or several variant structures incrementally by incorporating new classes one by one. The resulting hierarchies can be interactively exploited and refined using a graphic browser.

In summary, initially proposed for knowledge acquisition and discovery in the artificial intelligence field, FCA have recently found interesting applications in software engineering area as mentioned above. Concept lattice and related structures have been proposed as a framework for dealing with the construction and maintenance of class hierarchies. Within this framework, hierarchies are guaranteed to have zero redundancy, by factoring out commonalities, and to be consistent with specialization. Another advantage of the FCA theory is that it provides a clear and simple definition of the nature of generated hierarchy that does not depend on algorithm specifics or parameter tuning.

2.3 Mathematical Foundation of FCA

In the following we try to explain the basic concepts of formal concept analysis, which is a theory that has been developed since about 1980 by Wille (Wille 1982) and members of the Research Group on Formal Concept Analysis at the Technische Hochschule Darmstadt, and we shall do this with help of an example.

2.3.1 Basic Definitions and Theorems

Definition 1 A formal context is defined as a triple C = (O, A, R) where O and A are sets of objects[-] and attributes[-], respectively, and $R \subseteq O \times A$ is a binary (incidence) relation between O and A.

A context is typically represented in a tabular form as a cross table, whose rows are represented by the objects, whose columns are represented by attributes and whose cells are marked if and only if the incidence relation holds for the corresponding pair of object and attribute. Objects form a context share a set of common attributes and vice versa. As an example we present a context of geometric objects from Davies (Davies 1997). The different geometric shapes from a set O of objects, and some features of geometric shapes are collected in a set A of attributes. The incidence relation R is given by the cross table as shown in Table 2.1.

- Note: there are two total different meanings of object and attribute used in this thesis. In order to avoid confusion we give the usage of meanings of them here. **Object** – (1) when used in FCA, it means any entity that has attributes (properties) e.g., a person has attributes like height, weight, gender, etc. (2) When used in OOP, it is an instance of a class. The attributes of an object are defined by the instant variables of the class. So an object in FCA can be a class in OOP. **Attribute** – (1) when used in FCA it means a property of an entity. (2) when used in OOP it means an instant variable defined in a class. So an attribute of a class in OOP is also an attribute of the class object in FCA, but a method of a class in OOP that defines the class behavior is an attribute of a class object in FCA.

Table 2.1. A geometric attribute cross table

Objects\Attributes	*location*	*length*	*area*	*volume*	*center*	*radius*
Point	x					
Segment		x			x	
Curve		x				
Circle			x		x	x
Sphere				x	x	x
Square			x		x	
Cube				x	x	
Polygon			x			
Pyramid				x		
Prism				x		

An example of a formal context : geometric shapes

The cross table should be read in the following way: each x marks a pair being an element of the incidence relation R, e.g. (**Circle**, *center*) is marked because the object **Circle** carries the attribute *center*, where (**Polygon**, *center*) is not marked because normally polygon does not have a well-defined center. Thus (o, a) $\in$ R should be interpreted as "the object *o* carries the attribute *a*".

The central notion of the FCA is the formal concept. In order to define formal concept we need to present two derivation operators.

Definition 2 Let C = (O, A, R) be a formal context, $O_i \subseteq O$ and $A_i \subseteq A$. The common attributes of O_i are defined by $\alpha(O_i)$ (or O_i') = $\{a \in A_i \mid \forall o \in O_i: (o, a) \in R\}$, the common objects of A_i by $\omega(A_i)$ (or A_i') = $\{o \in O_i \mid \forall a \in A_i: (o, a) \in R\}$ where i is index, can be 1, 2, ...

We follow the notation used by Fischer in this thesis (Fischer 1998). The $\alpha(O_i)$ (or O_i') contains all attributes that are common to all objects in O_i, and $\omega(A_i)$ (or A_i') is the set of all objects that carry all the attributes of A_i. A concept is a pair of objects and attributes,

each synonymous with the other. The pair (O_i, A_i) is complete with respect to R if and only if the above two conditions are met. Notice $\alpha(\phi) = A$ and $\omega(\phi) = O$.

It is easy to prove that these derivation operators satisfy the following simple rules (for all $O_1, O_2, O_i \subseteq O$ and all $A_1, A_2, A_i \subseteq A$ where $i = 1, 2, \ldots$ and (O_i, A_i) is a formal concept, see below):

Lemma

(1) $O_1 \subseteq O_2 \Rightarrow O_2' \subseteq O_1'$; $\quad A_1 \subseteq A_2 \Rightarrow A_2' \subseteq A_1'$

(2) $O_i \subseteq O_i'' \;\&\; O_i' = O_i'''$; $\quad A_i = A_i'' \;\&\; A_i' = A_i'''$

$O_i \subseteq A_i' \Leftrightarrow A_i \subseteq O_i'$

Definition 3 The pair $x_i = (O_i, A_i)$ is called a formal concept if and only if $\alpha(O_i)$ (or O_i') $= A_i$ and $\omega(A_i)$ (or A_i') $= O_i$. $\pi_o(x_i) = O_i$ is called x_i's extent and $\pi_a(x_i) = A_i$ is called x_i's intent. The set of all concepts of the context C is denoted by G(C).

This property says that all objects of the concept carry all its attributes and that there is no other object in O carrying all attribute of the concept. Concepts can be imagined as maximal rectangles (modulo permutations of rows and columns) in the cross table. In other words when looking at the cross table in Table 2.1 this property can be seen if rectangles totally covered with crosses can be identified, e.g. the four cells associated with Circle, Sphere, Center and Radius constitute such a rectangle. If we ignore the sequence of the rows and columns we can identify even more concepts, e.g. ignoring the row Sphere and column Volume (or moving them to another place) we achieve another rectangle/concept, namely the cells associated with the objects Circle and Square, and the attributes Area and Center. Generally, we can say that formal concepts correspond to maximal rectangles of crosses in the formal context, after appropriate permutations of the rows and columns.

Frequently, formal concepts of the form ({o}", {o}') or ({a}', {a}"), where *o* and *a* are any object and attribute, respectively, and o ∈ O and a ∈ A, are of special interest because they are "generated" by a single object *o* from O or by a single attribute *a* from A, respectively. In such a case of one-element sets, we usually omit the set brackets, and write them as (o", o') and (a', a''), respectively. The former is called *object concept* and the later is called *attribute concept*. The concept ({pyramid, prism}, {volume}) (node x_4 in Figure 2.1 below) in Table 2.1 is an object concept generated by ({pyramid}", {pyramid}').

Concepts are partially ordered. A concept's extent includes the extents of its subconcepts and the intent of a concept includes the intents of its superconcepts.

Definition 4 Let C be a formal context defined above, $x_1 = (O_1, A_1)$, $x_2 = (O_2, A_2)$ where $x_1, x_2 \in G(C)$ (or $O_1, O_2 \subseteq O$ & $A_1, A_2 \subseteq A$). x_1 and x_2 are ordered by the subconcept-superconcept relation, $x_1 \leq x_2$ if and only if $O_1 \subseteq O_2$.

The intent-part follows by duality, i.e. $x_1 \leq x_2$ if and only if $A_2 \subseteq A_1$. A concept x_1 is a subconcept of x_2 if and only if the set of its objects is a subset of the objects of x_2 or equivalent expression is if and only if the set of its attributes is a superset of the attributes x_2. Actually, the set of all formal concepts of a formal context forms a concept lattice. Typically, given the formal context (O, A, R), the infimum (bottom) and supremum (top) of the concept lattice are formed by (ϕ, A) and (O, ϕ), respectively, but not always. As an immediate consequence of the preceding definitions we get that the strict order corresponds to strict inclusion of extents and intents, i.e. $x_1 < x_2$ if and only if $O_1 \subset O_2$ and $A_2 \subset A_1$.

The following basic theorem of FCA states the structure induced by the concepts of a formal context and their ordering is always a complete lattice and the infimum (or meet) and supremum (or join) can also be expressed by the common attributes and objects. The supremum of two concepts x_1 and x_2 is denoted by $x_1 \vee x_2$, and for a set of concepts G as $\vee G$. The infimum of the two concepts is denoted by $x_1 \wedge x_2$, and for a set of concepts G as $\wedge G$.

Theorem 1 Let C = (O, A, R) be a formal context. Then G(C) is a complete lattice, the concept lattice of C. Its infimum and supremum operations (for any set $H \subset G(C)$ of concepts) are given by: $\wedge_{i\in I}(O_i, A_i) = (\cap_{i\in I} O_i, \alpha(\omega(\cup_{i\in I} A_i)))$ and

$$\vee_{i\in I}(O_i, A_i) = (\omega(\alpha(\cup_{i\in I} O_i)), \cap_{i\in I} A_i).$$

This theorem says in order to compute the infimum (greatest common subconcept) of two concepts, their extents must be intersected and their intents joined; the later set of attributes, then must be enlarged in order to fit the object set of the infimum. Analogously, the supremum (smallest common superconcept) of two concepts is computed by intersecting the attributes and joining the objects.

The concept lattice can be considered as a graph, that is, a relation. What happens if we again apply FCA to this derived relation? It turns out that the concept lattice reproduces itself (Wille 1982). Thus concepts do not breed new concepts, there is no proliferation of virtual information. The concept lattice is sometimes also referred to as the Galois lattice because the couple of functions (α, ω) forms a Galois connection between O and A, and the Galois lattice G for the binary relation is the set of all complete pairs (Barbut & Monjartdet 1970). Hence, $\alpha o \omega$ and $\omega o \alpha$ are closure operators; In Theorem 1 their application maintains the "maximal rectangle" property of the resulting concepts.

Each attribute and object has a uniquely determined defining concept in the lattice which allows a sparse labeling of the lattice. The defining concepts can directly be calculated from the attribute or object, respectively, and need not to be searched in the lattice.

Definition 5 Let G(C) be a complete lattice. The defining concept of an attribute $a \in A$ (object $o \in O$) is the greatest (smallest) concept x such that $a \in \pi_a(x)$ ($o \in \pi_o(x)$) holds. It is denoted by $\mu(a)$ ($\sigma(o)$).

Theorem 2 In any concept lattice we have $\mu(a) = (\omega(\{a\}), \alpha(\omega(\{a\})))$ and $\sigma(o) = (\omega(\alpha(\{o\})), \alpha(\{a\}))$.

The proofs of both Theorem 1 and 2 can be found in the reference (Davery and Priestley 1990).

2.3.2 Graphic Representation of Concept Lattice -- Line Diagram

Because of the dualism between objects and attributes and the fact that data analysts or any other users of FCA are interested in investigating structures and relationships we need a representation of concepts that treats both objects and attributes alike. This representation is realized in a line diagram (a lattice diagram with reduced labeling). A line diagram (also called Hasse diagram, Wille 1984) is a graphic visualization of the concept lattice which allows the investigation and interpretation of relationships between concepts, objects and attributes. This includes object hierarchies, if they exist in the given formal context. A line diagram contains the relationships between objects and attributes, and thus is an equivalent representation of a formal context, i.e., it contains exactly the same information as the cross table. Also dependencies and relationships between attributes can be easily detected in a line diagram. The corresponding line diagram of Table 2.1 is shown in Figure 2.1 for the above context of geometric objects (modified from Davies 1997).

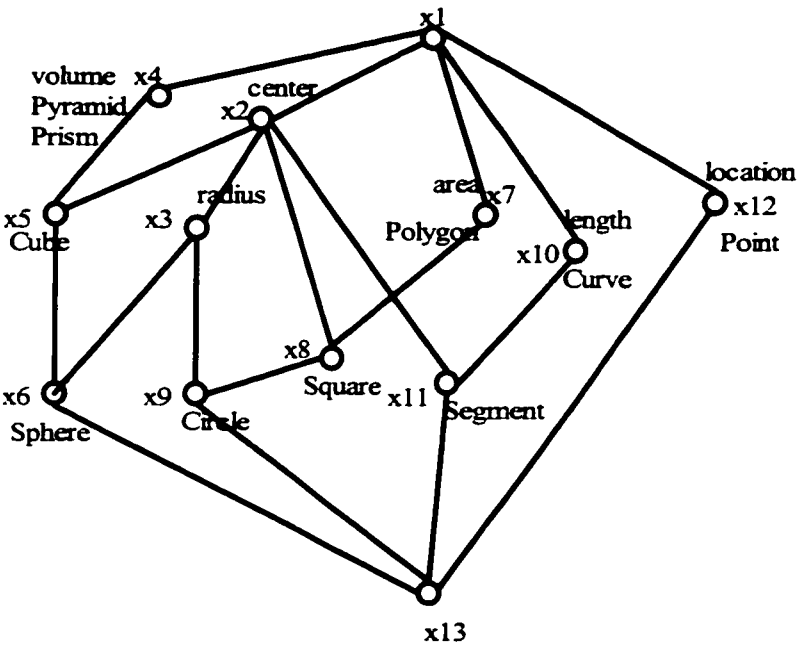

Figure 2.1 The concept lattice of the geometric context represented in line diagram

The graph consists of nodes that represent the concepts and edges connecting these nodes. Two nodes x_1 and x_2 are connected iff $x_1 \leq x_2$ and there is no concept x_3 with $x_1 \leq x_3 \leq x_2$. Although the concept lattice is a directed acyclic graph (DAG) the edges are not provided with arrowheads, instead the convention holds that the superconcept always appears above of all its subconcepts. For example, the line diagram shows that both node x_6 and x_9 are subconcepts of node x_3. As a difference to usual lattice diagrams the labeling in line diagram is reduced, i.e., each object and each attribute is the only entered once. So the nodes are not annotated by their complete extents and intents. Rather, attributes and objects propagate along the edges, as a kind of inheritance. Attributes propagate along the edge to the bottom of the diagram and dually objects propagate to the top of the diagram. Therefore the top element (the supremum of the context) is marked by (O, ϕ) if the O is the set of objects; the bottom element (infimum of the context) is marked by (ϕ, A) if the A is the set of attributes. Attributes names are always displayed slightly above the node and objects names are noted slightly below the respective node.

To read a line diagram we start at the object, attribute or concept you are interested in. Following all paths from this node to the top element, one visits all superconcepts of the selected concept. Collecting the attributes displayed along the paths one finds all attribute that the selected concept or object carries. Selecting a node and following all paths from this node to the bottom element (infimum of the context) one finds all subconcepts. If the selected node displays an attribute name all objects along these paths establish the set of objects carrying this attributes. Namely, the intent a concept x consists of all attributes, the names of which are attached to a circle (node x) which can be reached by an ascending line path from the node x (possibly including node x), while extent of x consists of all objects of x's subconcepts including x (i.e., can be reached by a descending line path from the node x). Thus line diagram displays relationships between objects, attributes and concepts in an easily perceivable way. Figure 2.1 reveals that Pyramid and Prism are equivalent objects. Of course one has to pay attention to the context. Concerning the given formal context Pyramid and Prism are equal because they carry exactly the same attribute, namely *volume*. Their equivalence can be seen in the line diagram by their appearance at the same concept node. The line diagram also displays object hierarchies and explicitly shows why some concepts are specialization of others. For example the line diagram shows, Square (node x_8) is a subconcept of Center (node x_2).

We will give another context and its corresponding line diagram to further illustrate how to read a line diagram (see Table 2.2 and Figure 2.2).

Table 2.2 a natural number context

Objects\Attributes	*even*	*odd*	*prime*	*square*	*cubic*
1		x		x	x
2	x		x		
3		x	x		
4	x			x	
5		x	x		
6	x				
7		x	x		
8	x				x
9		x		x	

Definition: even – is divisible by 2; odd – not is divisible by 2 without remainder; prime – is only divisible by 1 and itself; square – can be express as $n \times n = n^2$ for a natural number n; cubic -- can be express as $n \times n \times n = n^3$ for a natural number n.

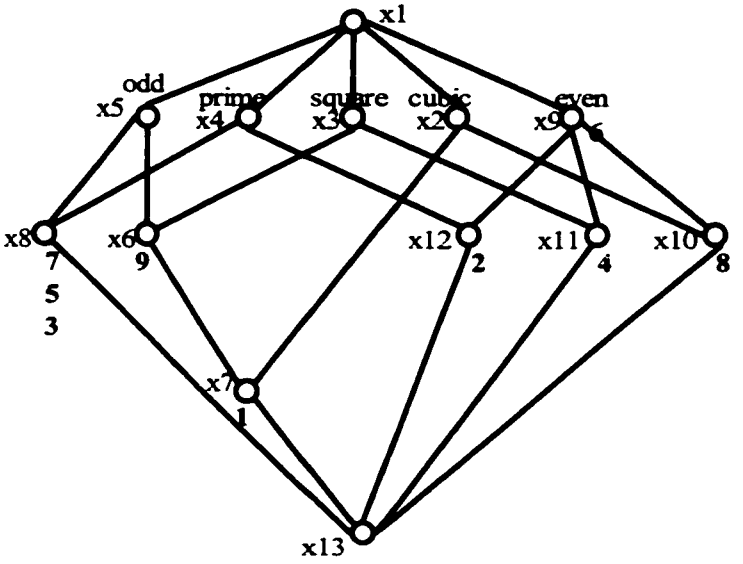

Figure 2.2 A line diagram of the natural number context

The above examples already demonstrate one possible interpretation of a concept lattice: it can be seen as a hierarchical conceptual clustering of the objects. Objects are grouped into sets, and the lattice structure imposes a taxonomy on these object sets. The original cross table can always be reconstructed from the concept lattice. Hence, a context cross table (i.e. relation) and its concept lattice are analogous to a function and its Fourier transform (which can also be reconstructed from each other): concept lattice is similar in spirit to a spectral analysis of continuous signals (Snelting 1996).

Another thing one can learn from line diagram is implication, e.g. any object that has a *radius* also carries the attribute *center*. This is because all subconcepts of the concept annotated with *radius* are also subconcepts of the node annotated with *center*. From Figure 2.2 we can see that {cubic, odd} $\Rightarrow$ {square} is true. In other words, the attributes *odd* and *even* together implies all other attributes and thus the infimum of the two corresponding attributes concepts is the smallest concept of the concept lattice. In dealing with relations between the attributes in the present context one should be aware that the implication may be true only in this context. As matter of fact, using some efficient algorithms that generating concept lattice and rules from a binary relation or even a database can help reveal implication and knowledge discovery (Godin, and Missaoui 1994).

2.3.3 Conceptual Scaling

Object-attribute-value relationships are frequently used data structure to code real-world problems. In this case, we have to use a many-valued context instead of one-valued context discussed above. A many-valued context is formally defined as a quadruple (O, A, V, R), where O, A, and V are sets whose elements are called objects, (many-valued) attributes and attributes values, respectively, and R is a ternary relation with $R \subseteq O \times A \times V$ such that $(o, a, v) \in R$ and $(o, a, w) \in R$ always implies $v = w$. An attribute of a many-valued context (O, A, V, R) may be considered as a partial map of O to V which suggests to write $a(o) = w$ rather than $(o, a, w) \in R$. The context (O, A, V, R) is called m-valued if V has cardinality m. One-valued contexts correspond to the formal context defined in previous section.

To construct concept lattice of a many-valued context (O, A, V, R) we need to transform it into a single-valued context (O, N, I) with the same objects as (O, A, V, R) whose extents can be thought of as the "meaningful" subsets of O. In general there is no

immediate, automatic way to derive the conceptual structures of data context which are based on object-attribute-value relationships. The first approach to transform these data contexts into concept lattice is called conceptual scaling. The concept lattice of the derived context is considered to be the conceptual structure of many-valued context. "Meaningful" refers to the interpretation of data which can only be by an expert of the field the data is from and never by the mathematician along (Prediger 97). This interpretation must always be purpose-oriented and should be founded on theoretical considerations.

The basic idea of conceptual scaling is to derive the context by conceptual scales. We assign to each many-valued attribute $a \in A$ a conceptual scale $S_a := (O_a, A_a, R_a)$ with $a(O) \subseteq O_a$. The choice of these scales is a matter of interpretation. The task is to select S_a in such way that it reflects the implicitly given structure of attribute values as well as the issues of data analysis.

The second step of concept scaling is to decide how the different many-valued attributes can be combined to describe concepts. The disjoint union of attribute sets often proves sufficient and we can thus restrict ourselves to looking at this so called plain conceptual scaling.

In other words, we can say the conceptual scaling yields a global view of the conceptual patterns of data stored in many-valued contexts by applying expert knowledge about the inherent structure of some or all attributes values. Sometimes, however, it is not the global view that is desired result but the answer to more specific questions. In this case, we already have a certain conception of relevant combinations of attributes, so it is not necessary to scale all attributes. Instead, we use these relevant combination of attributes to specify a limited terminology by which we can derive the context. This method is called logical scaling (for more details, see Prediger 97).

2.3.4 Generating Concept Lattice of a Given Context

There are several algorithms available to compute all the concepts from a given context, for example, for one-valued (C*, T*, I*) representing our class library (see chapter 3) and we construct a lattice of the concepts based on their sub-concept and super-concept relations through the greatest lower bound ($\wedge$) and the least upper bound ($\vee$) operations (for more details see the author's 60-510 survey report tilted "Formal Concept Analysis and Its Application in Software Engineering/Reuse" 1999). Among which, Ganter's algorithm is most efficient one (Ganter 1986). The cross table of a given context provides the input to the process of concept lattice formation. The resulting concept lattice is the class hierarchy. Each node of the concept lattice is a class, the parent classes of which are all concept nodes which are supeconcepts of the node. The top and bottom nodes of the lattice usually have an empty intent and empty extent, respectively. A concept will be labeled with attribute *a* if it is the largest concept having *a* in its intent. In other words all the concepts below the concept labeled with the *a* contains *a* in their intent set of attribute. Similarly, a concept is labeled with an object *o* if it is the smallest concept having object *o* in its extent.

We build a concept lattice using the superconcept-subconcept relationship through a line diagram in which each node is labeled with attributes of the set all concepts in the context.

Chapter 3 Methodology of Concept-Based Retrieval

3.1 Organizing a Class Library into a Formal Context

Large reuse libraries are valuable assets, however, the larger they grow, the harder it becomes to effectively manage them for reuse purpose. One of the major problems is to keep a high level overview of a library and extract appropriate components from the library. This requires better and structured organization of libraries and efficient retrieval algorithms than a flat file list of components. The concept-based approach is proposed to solve this problem.

3.1.1 Class and Its Behavior

Before indexing a class using either type information alone or type attached with information about ways of how methods access variables (updating variables, reading variables, or combination of the both, or doing nothing on the variables), let's first discuss some basic concepts of class to avoid confusion.

A class in OO languages defines its instance variables and related instance methods for a collection of objects having common attributes and behaviors (determined by its instance methods). Of course, a class in Java and Smalltalk can also defines its class variables and class methods (for example, the variables and methods with a key word: *static* in Java and Smalltalk). Just like objects, the class can have state, represented by the values of its class variables. Nevertheless, significant differences exist between classes and objects. Perhaps the most important difference is the way in which instance and class methods are invoked in Java: instance methods are (for the most part) dynamically bound, but class methods

are statically bound. (In three special cases, instance methods are not dynamically bound: invocation of private instance methods; invocation of *init* methods (constructors); and invocations with the *super* keyword.).

Classes can be defined as subclass and superclass relationship hierarchically via class inheritance. Both Java and Smalltalk support sharing data by all instances of a class and its subclasses. The shared data are stored in class variables of the class. A class variable can be used in any methods (including instance and class methods) defined in the class and in its subclasses. Changing the values of a class variable in one method changes it for all classes that share the variable. A class variable exists even when no instance of the class has been created yet. In contrast to a class variable, each instance of a class has its own copy of an instance variable defined by the class. The private memory of an instance (i.e., an object) is composed of instance variables. A variable defined in a superclass can't be redefined in a subclass. Subclasses can inherit the instant methods and redefine these instant methods from superclass if necessary, but can't redefine class methods although they can access these class methods. Similarly, subclass can access class variables defined in the subclass and its superclasses, but subclass can't access private (instant) variables defined in its superclasses. Any instance methods can change the values of class variables, therefore class variables are considered as global variables. Therefore, both class variables and class methods are used in restricted situations like defining utility methods and implementing special kinds of access policies for objects and primitive types stored in class variables. We will restrict ourselves to instance variables and instance methods in this thesis when we say class behavior (mainly defined by its instant methods) and a variable access pattern unless explicitly specified. Given a class, not only the instance methods defined in the class, but also the instance methods defined in its superclass are accessible to the class provided that these methods have a public or protected interface.

A class behavior is mainly defined by how its instant methods access its instant variables. The instant variables are used to store information and to transfer information within the class. Due to the fact that a class defines its instance variables and related instance methods for a collection of objects having common attributes and behaviors, a class behavior also represents object behavior of the class' instance.

3.1.2 Using Variable Access Pattern of a Method as Class Behavior

To get the information of a class behavior, one way used by Niu (Niu 1999) is that sending a test program constructed by user to the class and executing these methods of the class. As discussed in Chapter 1, there are many drawbacks of adopting this approach. In this thesis, we use another approach: using variable access pattern (denoted as p) of a method of a class to partially represent the class' behavior. This approach does not mean a complete coverage of the class' behavior. It is an attempt to reduce retrieval complexity and cost yet still maintain high recall and precision. However, we can always incorporate more information for a better and complete coverage of a class' behavior if there is such an interest. So our methodology is open and flexible.

We define the variable access patterns of the methods of a class as follows:

- *nRnW*: A method neither reads nor writes any variables.
- *nRaW*: A method does not read any variables but always writes some variable.
- *nRsW*: A method does not read any variables but sometimes writes some variable.
- *aRnW*: A method always reads some variable but does not write any variables.
- *aRaW*: A method always reads and writes some variable.
- *aRsW*: A method always reads some variable and sometimes writes some variable.
- *sRnW*: A method sometimes reads some variable but does not write any variables.

- *sRaW*: A method sometimes reads some variable and always writes some variable.
- *sRsW*: A method sometimes reads and writes some variable.

We characterize each class using the type information of variables/methods available in the class and the variable access patterns of these methods. Then, each class is represented as a *multiset* of the types of variables/methods and the access patterns of methods that are available in the class. This is because a class can have generally more than one variable or method of the same type and same access pattern. Let c be a class with n available variables/methods. Then,

Class c = *Multiset* $\{t_1{:}p_1, t_2{:}p_2, \ldots, t_n{:}p_n\}$

where t_i is the type and p_i is the access pattern of the i-th member in class c. Note that the set is a multiset that allows repetitions of elements.

3.1.3 A Class Library as a Multi-Valued Context

As discussed early in last chapter, a quadruple (O, A, V, R) is a many–valued context where O, A, and V are sets whose elements are objects, (many-valued) attributes and attributes values of the context, respectively, and R is a ternary relation with $R \subseteq O \times A \times V$ such that $(o, a, v) \in R$ means that the attribute a has the value v for the object o, and $(o, a, v) \in R$ and $(o, a, w) \in R$ always implies $v = w$. The context (O, A, V, R) is called m-valued if V has cardinality m. A many-valued context can also be represented as a table with rows of objects, columns of attributes and entries of attribute values.

Consider a class library L. Let

- C = {c | c is a class in L}
- T = {t | t is the type of a variable/method available in c and c is a class in L} $\cup$ {t^i | n variables/methods have the same type t and the same access pattern in a class c in L, t^i is i-th occurrence of t for i = 1, 2, …, n }

- P = {s | s = {p | p is the variable access patterns of an available variable/method of type t in c and c is a class in L} }. Note *s* is a set of access patterns of all variable/method in L.
- I = The ternary relation with $I \subseteq C \times T \times P$ defined as follows:

  (c, t, s) $\in$ I means that the class c has a variable/method of type t and the access pattern of the variable/method is p where p $\in$ s.

Then, the quadruple (C, T, P, I) forms a many-valued context. In general, we can view a given class library as a many-valued context. However, in practice, a class libray can be directly organized into a one-value context if each of all its attributes has just one value.

3.1.4 Converting into a One-Valued Context via Conceptual Scaling

In a many-valued context, if each attribute has only one value used for some property that an object may or may not have, then the context is a one-valued context. So a triple (O, A, R) is a one–valued context where O and A are sets whose elements are called objects and attributes, respectively, and R is a binary relation with $R \subseteq O \times A$ such that (*o, a*) $\in$ R means that the object *o* has the attribute *a*.

A many-valued context can be transformed to a one-valued context via a process called conceptual scaling discussed in the section 2.3.3 of the chapter 2, using scales for the attributes of the many-valued context.

From the many-valued context (C, T, P, I) that represent a class library, we derive a one-valued context (C*, T*, I*) using plain scaling with scales as follows:

- C* = C
- T* = { t:p | A variable/method of type t with the access pattern p is available in c and c is a class in L }
- I* = The binary relation with $I^* \subseteq C^* \times T^*$ defined as follows: (c, t:p) $\in$ I* if and only if (c, t, s) $\in$ I and p $\in$ s (which means that the class c has an available

variable/method of type t and the access pattern of the method is p. If it is a variable then access pattern p is equal to v, which indicates that it is a variable not a method)

3.2 Class Retrieval Based on the Concept Lattice

3.2.1 Browsing and Retrieving a Class Library

As we have seen above the class library can be eventually represented as a single-valued context (i.e., a binary relation) after a plain conceptual scaling. From this context we then constructed a concept lattice. Navigating the concept lattice of the class library actually does the search. Intuitively, user only needs to visit those nodes (concepts) to search the classes that meet his/her requirements. The user provides the composite types (type plus access pattern) of the variables/methods of a class/classes that are desirable to him/her. Library browsing and retrieval are closely related, but a clear distinction maybe made according to Mili et al (Mili, Mili and Mittermeir 1998). Retrieval focuses on extracting class components that satisfy a predefined matching criterion. Its main operation is thus the satisfaction check or matching. The criterion is usually given by an arbitrary user-defined search key or query which is matched against the candidates' indices. Retrieval supports a top-down design approach: the desired class component is first designed (or specified or selected reusable class components) and then looked up in the reuse library. Its main concern is thus precision: class components should not be retrieved unless they are absolutely relevant.

Browsing the library focuses on inspecting candidates for possible extraction, but without a predefined criterion. Its main operation is thus navigation which determines in what order the components are visited at all. Browsing supports a bottom-up design approach: the library is first inspected and then the system is designed (i.e., composed) to take the maximal advantage of the library. Its main concern is thus recall: class components should not be rejected unless they are absolutely irrelevant. Browsing usually works

stepwise and we denote the set of all class components which can be visited in the next step as the focus. In contrast to retrieval, it requires no search key but works on preprocessed, usually hierarchical navigation structure. The obvious although may not optimal way to compute such a structure is to order the class components (concepts from a given context) by FCA approach.

In this thesis we don't explicitly distinguish browsing and retrieving according to the above definitions. We redefine the browsing and retrieval as follows instead:

Retrieving is a process of satisfying the user query by extracting class components from a class library. The order of visiting a node (concept) is usually determined by the user's incremental selection of composite types, controlled the system not by the user.

Browsing is a process of looking around in a library without any user queries. The order of visiting a node (concept) is totally controlled by the user not by the system.

3.2.2 Approach to Concept-based Retrieval

In a class library, if there is a class component whose behavior exactly matches the desired behavior specified by the user through a query, then the retrieve engine will catch this class. Otherwise, the retrieve engine should yield those classes whose behavior are similar to the desired one, as these classes are more like to be easily tailored to behave as desired, or they can be subclassed to yield the desired behavior.

The class retrieval process will be based on the concept lattice for the one-valued context (C*, T*, I*) that represents a given class library. This process can be divided into following five phases:

- *Construction of a class library.* Reusable class components (maybe designed for reuse) are first selected from available sources.
- *Organizing the class library into a formal context.* Characterizing each class in the class library with type information plus variable access pattern (in short, composite types) of each available method in the class, i.e., viewing all the classes in the library as multi-sets of composite types. A context is organized in a tabular form as a cross table, whose rows are represented by the classes of the library, whose columns are represented by types and whose cells may have multi-values. If this is the case, we have to scale the context to convert it into a single-valued context, whose rows are still represented by the classes of the library, whose columns are represented by types plus variable access patterns, whose cells are marked if and only if the incidence relation holds for the corresponding pair of class and composite type. Classes form a context share a set of common composite types and vice versa.
- *Building the concept lattice based on the context.* We build a concept lattice using Ganter's algorithm (Ganter 1986) by superconcept-subconcept relation through the greatest low bound ($\wedge$) and the least upper bound ($\vee$) operations.
- *Formulating user query.* In order to use the retrieve system a user must be responsible for formulating a query that represents his/her need. The user formulates the query by "walking through" hierarchy structure through help of the system by specifying the concept he/she is interested in. Choosing one or more composite types (the desired behavior) available from the list displayed by the system will reduce the total numbers of available composite types. Each chosen composite type refines the query. Repeating the above steps eventually narrows search target to one or more concepts whose intents have all necessary composite types requested by the user. By formulating this query the user is saying that he/she is interested in one class or more classes that have the behavior like "int:aRnW", "String:aRnW", "void:nRaW".
- *Satisfying the user query (retrieval).* The set of classes matching a query (called a result, see chapter 4) can be retrieved by finding the meet of the greatest low bound of selected concepts.

The whole process of concept-based retrieval is schematically shown in Figure 3.1 below.

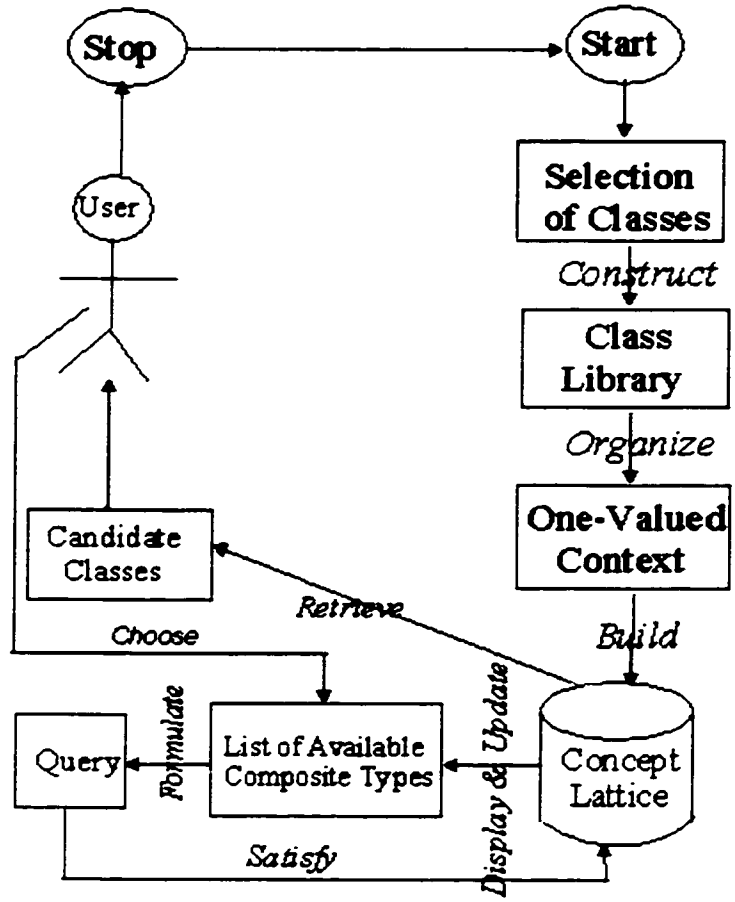

Figure 3.1 Process of concept-based retrieval

3.2.3 Procedures for Exact and Approximate Retrievals

The user searches for a class by providing the types plus the variable access patterns of the variables/methods that are available to the desired class. The system designed to handle concept lattice constructed from our class library context supports both browsing and retrieval functionality. The class retrieval is achieved through comparing and matching available composite types at one or more nodes with user's specific requirements (i.e. user query).

The user can also select this information incrementally from a list, displayed by the system, of available types plus access patterns. The list M of available types plus access patterns at a concept x is given as follows:

M = All attributes that label the concepts whose greatest lower bound with x is not the bottom concept.

Based on this information, the retrieval system searches the class library by navigating the concept lattice for the library. We can think of two kinds of retrievals: exact retrieval and approximate retrieval. Exact retrieval finds those classes that have variables/methods whose types plus their access patterns match exactly with the query, i.e., types plus access patterns of the variables/methods of the desired class. If no exactly matched class is found, classes that are not exactly matched but somewhat related to the desired class could be useful for reuse purpose. Approximate retrieval finds those classes that partially match with the query.

Let Q be a search query, then $Q \subseteq T^*$, i.e., $Q = \{t_1{:}p_1, t_2{:}p_2, ..., t_k{:}p_k\}$ where $t_i{:}p_i \in T^*$ in the one-valued context (C^*, T^*, I^*) for a class library. Let $(E, F) = \vee_{q \in Q}$ where maximal(q) is the largest lattice concept. Then $Q \subseteq F$ and $o \in E \Rightarrow \{o\}' \in F$. The exact retrieval process is done as follows:

1) Locate the concepts labeled with attributes $t_1{:}p_1$, $t_2{:}p_2$, ..., and $t_k{:}p_k$, respectively.

2) Locate the greatest lower bound of all those concepts by following the concept hierarchy.

3) Retrieve those classes that have *k* accessible members from the extent of the resulting concept.

Two types of approximate retrieval can be done as follows:

- In order to retrieve classes that have *l* fewer variables/methods than the desired class but whose types and access patterns match the desired class: *Construct all*

the subsets that have (k - l) elements from the original query set Q. Using each subset as a query, perform the exact retrieval process.

- In order to retrieve the classes that have *m* more variables/methods than the desired class: *Retrieve those classes that have (k + m) members from extent of the resulting concept.*

3.2.4 Retrieving Classes Regardless of Argument Order

The order of arguments of the methods in the desired class is often not important at the stage of searching reusable classes. The user is likely to search regardless of the arguments order of the methods.

Similar to Park's previous work (Park 1999) we can also easily achieve retrieving classes regardless of arguments order by again using the notion of set types. we define a set type as an extended type that captures all the methods of the same type by ignoring the order of arguments. The regular type of a method consists of a tuple of parameters' type (t_1, t_2, ...) and the return type (t_r). In set types, we represent a multiset of parameters' types and the return type.

For example, the set type $\{t_1, t_2, t_r\} \rightarrow t_r$ contains the regular types $(t_1 \times t_1 \times t_2) \rightarrow t_r$, $(t_1 \times t_2 \times t_1) \rightarrow t_r$ and $(t_2 \times t_1 \times t_1) \rightarrow t_r$. We then represent a class component as multiset of set types of the variables and methods that are accessible to the class. Suppose that the user wants to retrieve a class that has a methods with two arguments (but the arguments order is not important) such that the type of the argument is t_1 and the other arguments types is t_2, and the return type is t_r. The user provides $\{t_1, t_2\} \rightarrow t_r$ as the type information of the desired method.

In practice, of course we will achieve the above purpose differently in our retrieve system. If we incorporate the parameter types of each method of each class in a class library into the single-valued context, then the parameter types of the method will be attached to its return type t. Then a composite type in the context will consist of three parts: (return) type t, parameter types $t_{arg:1}$, ..., $t_{arg:n}$, and variable access pattern p. In this case, the system will display a list of all the three-part composite types that are available for the user to choose in the class library. Since the user is not allowed to enter an arbitrary composite type to the user query but choose an item from the list, the problem of argument order of a method will be completely eliminated in our CBRS system.

3.3 An Example of Model Class Library

3.3.1 Representing the Class Library Using UML

Consider, for example, an "imaginary" class library L consisting of seven classes as shown in Figure 3.2.

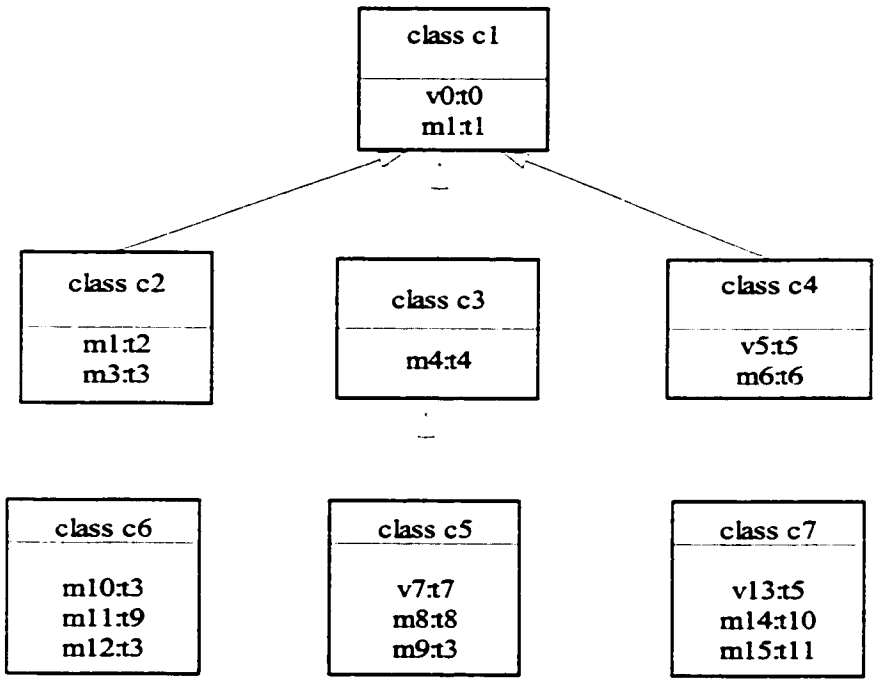

Figure 3.2 A sample class library L

3.3.2 Variable Accessing Patterns of Methods

The class diagrams in the Figure 3.2 is depicted in UML notation. The method m_1 in class c_2 overrides the method m_1 in the class c_1, but c_1's subclass c_2 can access m_1:t_2 but can not directly access method m_1:t_1 in class c_1 (c_2 may access m_1:t_1 through using keyword *super* in SmallTalk and Java, for example if there is a such need). There are total 12 distinct types of variables/methods within all classes of the class library. Assume that the accessing behaviors of all the methods in the class library L are as summarized in Table 3.1.

Methods	Accessing Behaviors of Methods
m_1 in C_1	never read & always write from/to a variable (nRaW)
m_1 in C_2	never read & always write from/to a variable (nRaW)
m_3 in C_2	always read & never write from/to a variable (aRnW)
m_4 in C_3	always read & never write from/to a variable (aRnW)
m_6 in C_4	always read & always write from/to a variable (aRaW)
m_8 in C_5	never read & never write from/to a variable (nRnW)
m_9 in C_5	never read & always write from/to a variable (nRaW)
m_{10} in C_6	always read & always write from/to a variable (aRaW)
m_{11} in C_6	never read & always write from/to a variable (nRaW)
m_{12} in C_6	sometimes read & always write from/to a variable (sRaW)
m_{14} in C_7	always read & never write from/to a variable (aRnW)
m_{15} in C_7	always read & always write from/to a variable (aRaW)

Table 3.1 Access bahavior of methods in the classes in L

3.3.3 Organizing a Single-Valued Context

Based on the information given in Figure 3.2 and Table 3.1, all the classes in the library can be represented as multi-sets as follows:

$c_1 = \{t_0:v, t_1:nRaW\}$

$c_2 = \{t_0:v, t_1:nRaW, t_2:nRaW, t_3:aRnW\}$

$c_3 = \{t_0:v, t_1:nRaW, t_4:aRnW\}$

$c_4 = \{t_0:v, t_1:nRaW, t_5:v, t_6:aRaW\}$

c_5 = {t_0:v, t_1:nRaW, t_4:aRnW, t_7:v, t_8:nRnW, t_3:nRaW}

c_6 = {t_3:aRaW, t_9:nRaW, t_3:sRaW }

c_7 = {t_5:v, t_{10}:aRnW, t_{11}:aRaW}.

The class library L can be viewed as the following many-valued context (C, T, P, I) where

- C = {c_1, c_2, c_3, c_4, c_5, c_6, c_7}
- T = {t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}}
- P = {{nRnW}, {nRaW}, {aRnW}, {aRaW}, {sRaW}, {v}}
- I = The ternary relation with I $\subseteq$ C × T × P defined in Table 3.2:

I	t_0	t_1	t_2	t_3	t_4	t_5
c_1	*{v}*	*{nRaW}*				
c_2	*{v}*	*{nRaW}*	*{nRaW}*	*{aRnW}*		
c_3	*{v}*	*{nRaW}*			*{aRnW}*	
c_4	*{v}*	*{nRaW}*				*{v}*
c_5	*{v}*	*{nRaW}*		*{nRaW}*	*{aRnW}*	
c_6				*{aRaW, sRaW}*		
c_7						*{v}*

I	t_6	t_7	t_8	t_9	t_{10}	t_{11}
c_1						
c_2						
c_3						
c_4	*{aRaW}*					
c_5		*{v}*	*{nRnW}*			
c_6				*{nRaW}*		
c_7					*{aRnW}*	*{aRaW}*

Table 3.2 The many-valued context for L

Note that the class c_6's two methods m_{10} and m_{12} have the same type t_3, but with different access patterns aRaW and sRaW, respectively.

The one-valued context (C*, T*, I*) is derived form the many-valued context after plain conceptual scaling with scales for the attributes where

- $C^* = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7\}$
- $T^* = \{ t_0$:v, t_1:nRaW, t_2:nRaW, t_3:aRnW, t_3:nRaW, t_3:aRaW, t_3:sRaW, t_4:aRnW, t_5:v, t_6:aRaW, t_7:v, t_8:nRnW, t_9:nRaW, tv_{10}:aRnW, t_{11}:aRnW$\}$
- I* = The binary relation with $I^* \subseteq C^* \times T^*$ defined in Table 3.3

I*	t_0:v	t_1:nRaW	t_2:nRaW	t_3:aRnW	t_3:nRaW	t_3:aRaW	t_3:sRaW
c_1	x	x					
c_2	x	x	x	x			
c_3	x	x					
c_4	x	x					
c_5	x	x			x		
c_6						x	x
c_7							

I*	t_4:aRnW	t_5:v	t_6:aRaW	t_7:v	t_8:nRnW	t_9: nRaW	t_{10}:aRnW	t_{11}:aRnW
c_1								
c_2								
c_3	x							
c_4		x	x					
c_5	x			x	x			
c_6						x		
c_7		x					x	x

Table 3.3 The one-valued context for L

3.3.4 Building Concept Lattice

As discusses in chapter 2 we construct the concept lattice of the library based on sub-concept and super-concept relationships through the greatest lower bound ($\wedge$) and the least upper bound ($\vee$) operations.

All concepts and their subconcept-superconcept relations for this one-valued context (C*, T*, I*) can be computed as follows:

Concept Top $X_0 = (\{c_1, c_2, c_3, c_4, c_5, c_6, c_7\}, \{\})$

Concept $X_1 = (\{c_1, c_2, c_3, c_4, c_5\}, \{t_0{:}v, t_1{:}nRaW\})$

Concept $X_2 = (\{c_2\}, \{t_0{:}v, t_1{:}nRaW, t_2{:}nRaW, t_3{:}aRnW\})$

Concept $X_3 = (\{c_3, c_5\}, \{t_0{:}v, t_1{:}nRaW, t_4{:}aRnW\})$

Concept $X_4 = (\{c_4\}, \{t_0{:}v, t_1{:}nRaW, t_5{:}v, t_6{:}aRaW\})$

Concept $X_5 = (\{c_5\}, \{t_0{:}v, t_1{:}nRaW, t_3{:}nRaW, t_4{:}aRnW, t_7{:}v, t_8{:}nRnW\})$

Concept $X_6 = (\{c_6\}, \{t_3{:}aRaW, t_3{:}sRaW, t_9{:}nRaW\})$

Concept $X7 = (\{c_7\}, \{t_5{:}v, t_{10}{:}aRnW, t_{11}{:}aRaW\})$

Concept Bottom $X_8 = (\{\}, \{t_0{:}v, t_1{:}nRaW, t_2{:}nRaW, t_3{:}aRnW, t_3{:}nRaW, t_3{:}aRaW, t_3{:}sRaW, t_4{:}aRnW, t_5{:}v, t_6{:}aRaW, t_7{:}v, t_8{:}nRnW, t_9{:}nRaW, t_{10}{:}aRnW, t_{11}{:}aRaW\})$

$X_1 \vee X_2 = X_1 \vee X_4 = X1 \vee X_3 = X_1$

$X_6 \vee X_7 = X_1 \vee X_0 = X_0$

$X_3 \vee X_5 = X_3$

$X_6 \vee X_8 = X_6$, $X_7 \vee X_8 = X_7$

The resulting concept lattice is depicted as a line diagram in Figure 3.3.

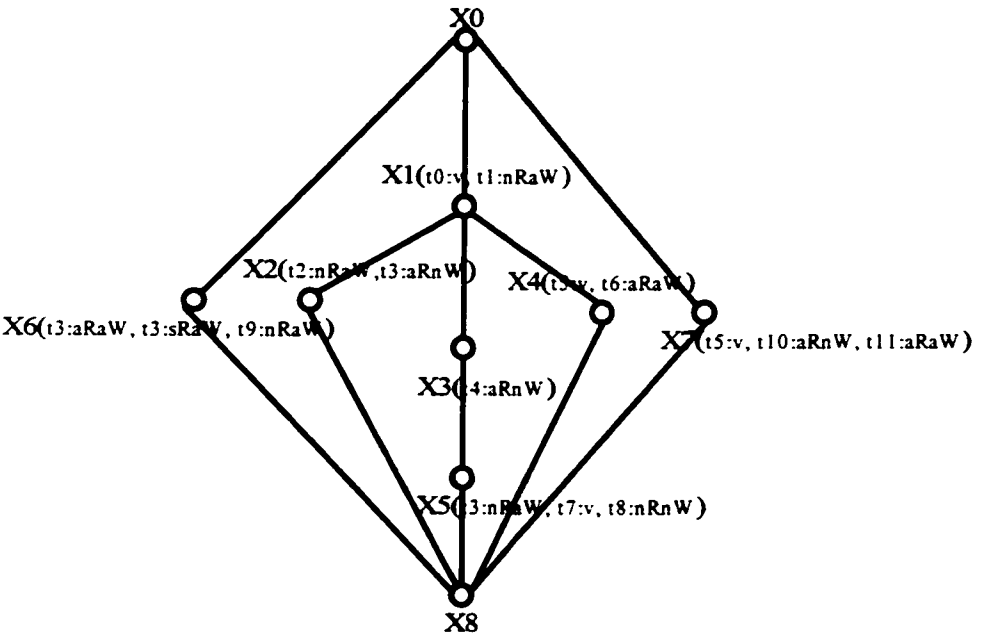

Figure 3.3 Concept lattice for library L

3.3.5 Formulating Queries and Retrieving Classes

Initially, all the fifteen types plus access patterns (composite types) are available for user to choose. In other words, all seven classes are available for retrieval at top concept X_0 as the system is default to the top concept. If the user wants to retrieve all the classes in the class library, he/she must submit a blank query at this point. If the user decides to choose anything from the available composite types, the system will be directed to appropriate concept whose intent has the composite types the user wants. In contrast, no types plus access patterns are available at concept X_8 if the system is directed to the bottom concept.

Consider, for example, the available types plus access patterns to select at the concept X_1 after selecting t_0:v and t_1:nRaW are {t_0:v, t_1:nRaW, t_2:nRaW, t_3:aRnW, t_3:nRaW, t_4:aRnW, t_5:v, t_6:aRaW, t_7:v, t_8:nWnR}.

Suppose that user so far has selected t_0:v, t_1:nRaW and t_3:aRnW (the system is at the concept X_2) and wants to select third method (always retrieving values from variables &

updating variables) of type t_3. Since t_3:aRaW is not available at this point, we can conclude that there is no class that has methods of composite types t_0:v, t_1:nRaW, t_3:aRnW and t_3:aRaW.

Consider the query Q_1 = {t_0:v, t_1:nRaW, t_5:v, t_6:aRaW}. Obviously, Q_1 is a subset of T* as defined above in the one-valued context. The exact retrieval system finds the concepts X_1 and X_4 that are labeled with {t_0:v, t_1:nRaW} and {t_5:v, t_6:aRaW}, respectively, and finds the concept X_4 as the greatest lower bound between X_1 and X_4. It then retrieves class c_4 that is extent of the concept X_4 because the class c_4 has all the required members in the query Q_1.

For the query Q_2 = {t_0:v, t_1:nRaW, t_4:aRnW, t_8:nRnW}, the exact retrieval system finds concept X_5 as greatest low bound of the concepts labeled with {t_3:nRaW, t_4:aRnW, t_7:v, t_8:nRnW} but there is no class with four required members in X_5. Class c_5 has six accessible members. If the user wants to see any class with all the six members including {t_0:v, t_1:nRaW, t_4:aRnW, t_8:nRnW}, then retrieval system will retrieve class c_5.

Consider the query Q_3 = {t_0:v, t_1:nRaW, t_2:nRaW, t_3:aRnW, t_5:v, t_6:aRaW}. The retrieval system finds the concepts X_1, X_2 and X_4 that are labeled with {t_0:v, t_1:nRaW}, {t_2:nRaW, t_3:aRnW} and {t_5:v, t_6:aRaW}, respectively, and finds the concept bottom as the greatest lower bound among X_1, X_2 and X_4. It then retrieves no class because the extent of the concept bottom X_8 has empty set.

Consider the query Q_2 = {t_0:v, t_1:nRaW, t_4:aRnW, t_8:nRnW} again, suppose the user wants to see any class with first three members from {t_0:v, t_1:nRaW, t_4:aRnW, t_8:nRnW}. The approximate retrieval system will search using all possible subqueries, i.e. {t_0:v, t_1:nRaW, t_4:aRnW}, {t_0:v, t_1:nRaW, t_8:nRnW}, {t_0:v, t_4:aRnW, t_8:nRnW} and {t_1:nRaW,

t_4:aRnW, t_8:nRnW}, and finally retrieve class c_3 containing members of composite types {t_0:v, t_1:nRaW and t_4:aRnW}.

3.4 Two Small-Scale Class Libraries

The above "imaginary" class library is used to successfully demonstrate that we could use principles of FCA to organize our library into a hierarchical structure, and precise class retrieval could be made easily based on the lattice structure. In order to further prove that our methodology works perfectly, we will present two real small-scale class libraries which will be analyzed and treated by the FCA approach. All the classes in the libraries are regular Java classes, they can be compiled and executed by Java compiler and Java Virtual Machine (JVM), respectively.

3.4.1 A Real Class Library 1

The first class library consists of eight Java classes: **Point, Circle, Sphere, Cube, Circle, Employee, Boss,** and **CommissionWorker** (**CWorker** for short in the context). All the members (including fields and methods) of each class and the relationships among classes are in details depicted by using UML, as shown in Figure 3.4 below (**Cube** and **Sphere** are two direct subclasses of **Point**, respectively, which are not drawn due to space limitation). The access modifiers "+", "#", and "-" in front of a variable/method in UML indicates that the variable/method are public, protected, and private, respectively.

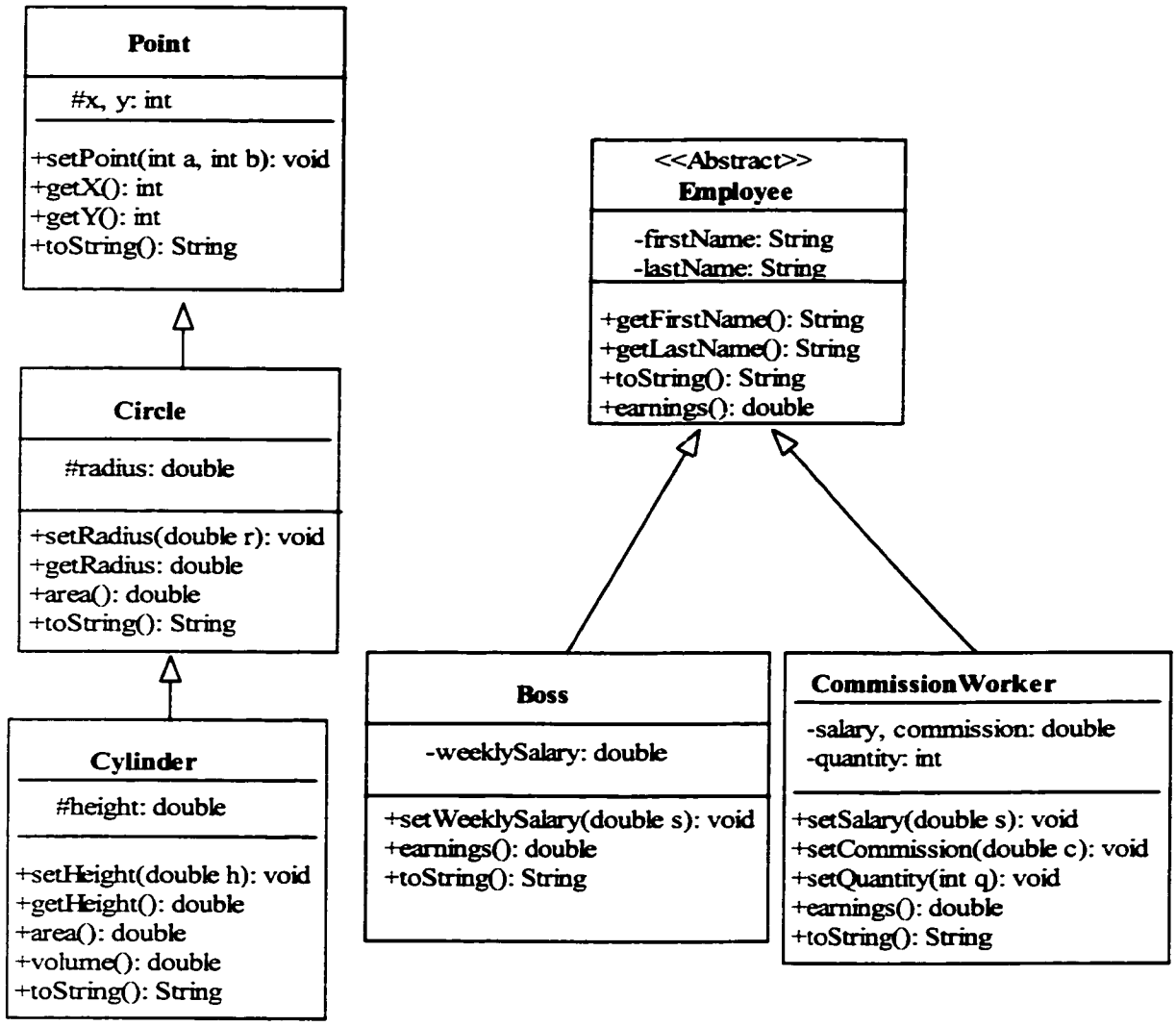

Figure 3.4 Class diagram of the class library 1

The corresponding one-valued context for this library is represented as a cross table as shown in Table 3.4 below. Note that we have switched the positions of object set and attribute set in this context to save space, i.e., in this single-valued context, its columns are represented by the classes of the library, its rows are represented by types plus variable access patterns (in short, composite types), whose cells are marked if and only if the incidence relation holds for the corresponding pair of class and composite type. This is also true to the class library 2 which will be discussed next.

Another feature that is worthy of notice is that all the types of variables/methods in the class library are primitive types of Java except void. This does not mean that this methodology only allows primitive data type. As matter of fact, user defined type (class type) is absolutely permitted during construction of single-valued context.

	Point	Circle	Cylinder	Sphere	Cube	Employee	Boss	CWorker
void[1]:nRaW	x	x	x	x	x			
int[1]:aRnW	x	x	x	x	x			
int[2]:aRnW	x	x	x	x	x			
String[1]:aRnW	x	x	x	x	x			
void[2]:nRaW		x	x					
double[1]:aRnW		x	x					
double[2]:aRnW		x	x					
String[2]:aRnW		x	x					
void[3]:nRaW			x					
double[3]:aRnW			x					
double[4]:aRnW			x					
double[5]:aRnW			x					
String[3]:aRnW			x					
void[4]:nRaW				x				
double[6]:aRnW				x				
double[7]:aRnW				x				
double[8]:aRnW				x				
String[4]:aRnW				x				
void[5]:nRaW					x			
double[9]:aRnW					x			
double[10]:aRnW					x			
double[11]:aRnW					x			
String[5]:aRnW					x			
String[6]:aRnW						x	x	x
String[7]:aRnW						x	x	x
String[8]:aRnW						x	x	x
double[12]:aRnW						x	x	x
void[6]:nRaW							x	
double[13]:aRnW							x	
String[9]:aRnW							x	
void[7]:nRaW								x
void[8]:nRaW								x
void[9]:nRaW								x
double[14]:aRnW								x
String[10]:aRnW								x

Table 3.4 One-valued context of class library 1

Based on the one-valued context in the Table 3.4, we use Ganter's algorithm (Ganter 1986) to calculate all the concepts of the lattice. Its corresponding concept lattice is schematically shown in Figure 3.5.

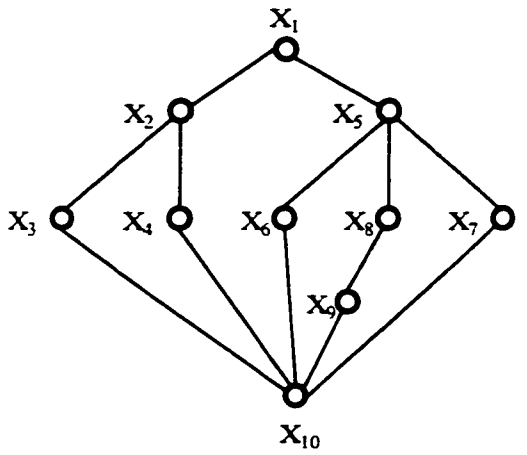

Figure 3.5 Concept lattice of the class library 1

To keep line diagram neat, we didn't label each concept with class(es) (object set) and composite types (attribute set). X_1 and X_{10} are top and bottom concepts, respectively. If we use C* and T* to denote the set of all the classes and the set of all the composite types in the class library, respectively, then the concepts of X_1 and X_{10} can be expressed as (C*, {}) and ({}, T*), respectively. The following is a list of labels for all the rests of the concepts in Figure 3.5:

Object Sets	**Attribute Sets**
X_2: {Employee}	{String[6]:aRnW, String[7]:aRnW, String[8]:aRnW, double[12]:aRnW}
X_3: {CWorker}	{void[7]:nRaW, void[8]:nRaW, void[9]:nRaW, double[14]:aRnW, String[10]:aRnW}
X_4: {Boss}	{void[6]:nRaW, double[13]:aRnW, String[9]:aRnW}
X_5: {Point}	{void[1]:nRaW, int[1]:aRnW, int[2]:aRnW, String[1]:aRnW}
X_6: {Cube}	{void[5]:nRaW, double[9]:aRnW, double[10]:aRnW, double[11]:aRnW, String[5]:aRnW}
X_7: {Sphere}	{void[4]:nRaW, double[6]:aRnW, double[7]:aRnW, double[8]:aRnW, String[4]:aRnW}
X_8: {Circle}	{void[2]:nRaW, double[1]:aRnW, double[2]:aRnW, String[2]:aRnW}
X_9: {Cylinder}	{void[3]:nRaW, double[3]:aRnW, double[4]:aRnW, double[5]:aRnW, String[3]:aRnW}

Now we can list each concept with its extent and intent, respectively, as follows:

Concept X_1 = (C*, {}); Concept X_{10} = ({}, T*)

Concept X_2 = ({Employee, CWorker, Boss}, {String^6:aRnW, String^7:aRnW, String^8:aRnW, double^{12}:aRnW})

Concept X_3 = ({CWorker}, {void^7:nRaW, void^8:nRaW, void^9:nRaW, double^{14}:aRnW, String^{10}:aRnW, String^6:aRnW, String^7:aRnW, String^8:aRnW, double^{12}:aRnW })

Concept X_4 = ({Boss}, {void^6:nRaW, double^{13}:aRnW, String^9:aRnW, String^6:aRnW, String^7:aRnW, String^8:aRnW, double^{12}:aRnW })

Concept X_5 = ({Point, Cube, Sphere, Circle, Cylinder}, {void^1:nRaW, int^1:aRnW, int^2:aRnW, String^1:aRnW})

Concept X_6 = ({Cube}, {void^5:nRaW, double^9:aRnW, double^{10}:aRnW, double^{11}:aRnW, String^5:aRnW, void^1:nRaW, int^1:aRnW, int^2:aRnW, String^1:aRnW})

Concept X_7 = ({Sphere}, {void^4:nRaW, double^6:aRnW, double^7:aRnW, double^8:aRnW, String^4:aRnW, void^1:nRaW, int^1:aRnW, int^2:aRnW, String^1:aRnW})

Concept X_8 = ({Circle, Cylinder}, {void^2:nRaW, double^1:aRnW, double^2:aRnW, String^2:aRnW, void^1:nRaW, int^1:aRnW, int^2:aRnW, String^1:aRnW})

Concept X_9 = ({Cylinder}, {void^3:nRaW, double^3:aRnW, double^4:aRnW, double^5:aRnW, String^3:aRnW, void^2:nRaW, double^1:aRnW, double^2:aRnW, String^2:aRnW, void^1:nRaW, int^1:aRnW, int^2:aRnW, String^1:aRnW})

Following the standard procedures described in section 3.2.3, the user can retrieve any classes he/she wants from this class library by simply formulating the query through a list (a **JList** data structure in Java term is used for the prototype system) of available composite types and satisfying the query to the retrieve system.

3.4.2 Another Real Class Library 2

The second class library also consists of eight Java classes: **CGrid, PrintCGrid, BorderedPrintCGrid, Point, CGObject, CGPoint, CGText,** and **CGBox**. The **Point** class is different in this class library from the one in previous class library. All the members (including fields and methods) of each class and the relationships among classes are in details depicted by using UML, as shown in Figure 3.6 below.

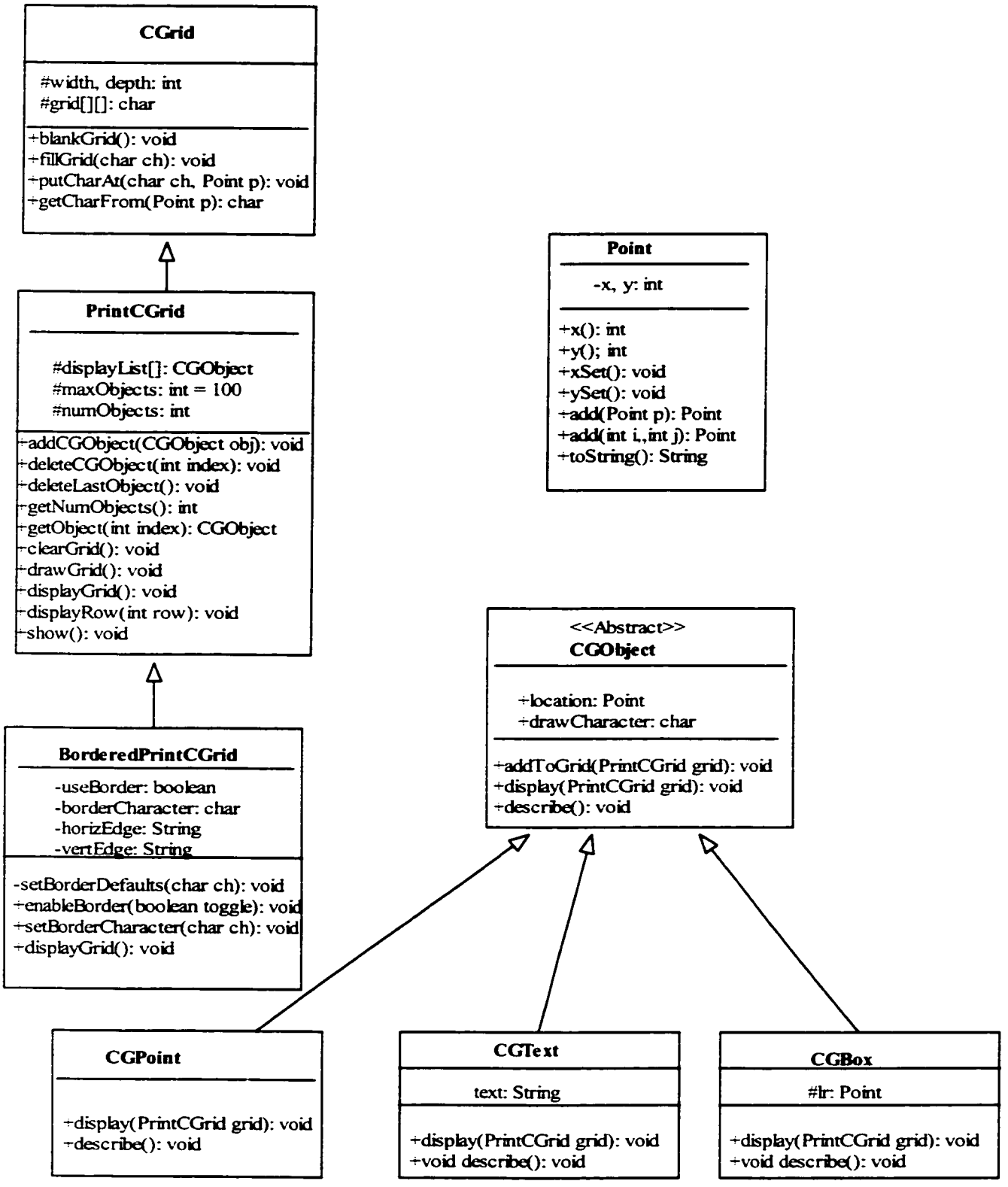

Figure 3.6 Class diagram of class library 2

	Point	CGObject	CGrid	PrintCGrid	BorderedPrintCGrid	CGPoint	CGText	CGBox
int[1]:aRnW	x							
int[2]:aRnW	x							
void[1]:nRaW	x							
void[2]:nRaW	x							
Point[1]:aRnW	x							
Point[2]:aRnW	x							
String[1]:aRnW	x							
Point[3]:v		x				x	x	x
char[1]:v		x				x	x	x
void[3]:		x				x	x	x
void[4]:		x				x	x	x
void[5]:		x				x	x	x
int[3]:v			x	x	x			
int[4]:v			x	x	x			
char[2]:v			x	x	x			
void[6]:nRaW			x	x	x			
void[7]:nRaW			x	x	x			
void[8]:nRaW			x	x	x			
char[3]:aRnW			x	x	x			
CGObject[1]:v				x	x			
int[5]:v				x	x			
int[6]:v				x	x			
void[9]:aRaW				x	x			
void[10]:aRaW				x	x			
void[11]:aRaW				x	x			
int[7]:aRnW				x	x			
CGObject[2]:aRnW				x	x			
void[12]:nRaW				x	x			
void[13]:aRaW				x	x			
void[14]:aRnW				x	x			
void[15]:aRnW				x	x			
void[16]:aRnW				x	x			
void[17]:nRaW					x			
void[18]:nRaW					x			
void[19]:aRaW					x			
void[20]:aRnW					x			
void[21]:aRaW						x		
void[22]:aRnW						x		
Point[4]:v								x
void[23]:aRaW								x
void[24]:aRnW								x
String[2]:v							x	
void[25]:aRaW							x	
void[26]:aRnW							x	

Black - Public (or Default) Blue - Protected Red - Private Magenta - Abstract

Table 3.5 One-valued context of class library 2

The corresponding one-valued context for this library is represented as a cross table as shown in Table 3.5 above. Again we have switched the positions of object set and attribute set in this context to save space, i.e., in this single-valued context, its columns are represented by the classes of the library, its rows are represented by types plus variable access patterns (in short, composite types), whose cells are marked if and only if the incidence relation holds for the corresponding pair of class and composite type.

Another prominent feature is that not only primitive types of Java but also user defined type (class type) appeared in the context. This demonstrates our methodology can handle both primitive data types and class types during construction of single-valued context and retrieving of classes.

Based on the one-valued context in the Table 3.5, we again use Ganter's algorithm to calculate all the concepts of the lattice. Its corresponding concept lattice without labeling is schematically shown in Figure 3.7 which is different from that of Figure 3.5. Obviously, concept lattice structure heavily relies on the content of the context. The different context results in different concept lattice.

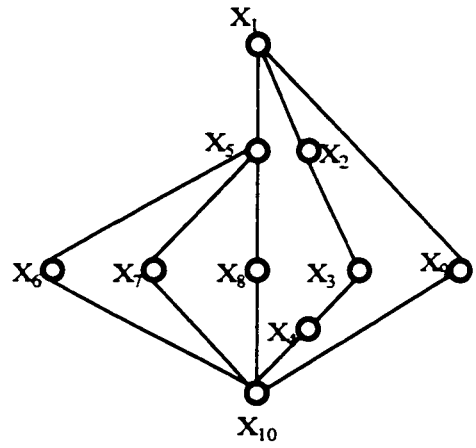

Figure 3.7 Concept lattice of class library 2

X_1 and X_{10} are top and bottom concepts, respectively. If we use C* and T* to denote the set of all the classes and the set of all the composite types in the class library, respectively, then the concepts of X_1 and X_{10} can be expressed as (C*, {}) and ({}, T*), respectively. The following is a list of labels for all the rests of the concepts in Figure 3.7.

Object Sets **Attribute Sets**

X_2: {CGrid} {int[3]:v, int[4]:v, char[2]:v, void[6]:nRaW, void[7]:nRaW, void[8]:nRaW, char[3]:aRnW}

X_3: {PrintCGrid} {CGObject[1]:v, int[5]:v, int[6]:v ,void[9]:aRaW, void[10]:aRaW, void[11]:aRaW, int[7]:aRnW, CGObject[2]:aRnW, void[12]:nRaW, void[13]:aRaW, void[14]:aRnW, void[15]:aRnW, void[16]:aRnW}

X_4: {BorderedPrintCGrid} {void[17]:nRaW, void[18]:nRaW, void[19]:aRaW, void[20]:aRnW}

X_5: {CGObject} {Point[3]:v, char[1]:v, void[3]:, void[4]:, void[5]:}

X_6: {CGText} {String[2]:v, void[25]:aRaW, void[26]:aRnW}

X_7: {CGBox} {Point[4]:v, void[23]:aRaW, void[24]:aRnW}

X_8: {CGPoint} {void[21]:aRaW, void[22]:aRnW}

X_9: {Point} {int[1]:aRnW, int[2]:aRnW, void[1]:nRaW, void[2]:nRaW, Point[1]:aRnW, Point[2]:aRnW, String[1]:aRnW}

Now we can list each concept with its extent and intent, respectively, as follows:

Concept X_1 = (C*, {}); Concept X_{10} = ({}, T*)

Concept X_2 = ({CGrid, PrintCGrid, BorderedPrintCGrid}, {int[3]:v, int[4]:v, char[2]:v, void[6]:nRaW, void[7]:nRaW, void[8]:nRaW, char[3]:aRnW})

Concept X_3 = ({PrintCGrid, BorderedPrintCGrid}, {CGObject[1]:v, int[5]:v, int[6]:v ,void[9]:aRaW, void[10]:aRaW, void[11]:aRaW, int[7]:aRnW,CGObject[2]:aRnW, void[12]:nRaW, void[13]:aRaW, void[14]:aRnW, void[15]:aRnW, void[16]:aRnW, int[3]:v, int[4]:v, char[2]:v, void[6]:nRaW, void[7]:nRaW, void[8]:nRaW, char[3]:aRnW})

Concept X_4 = ({BorderedPrintCGrid} {void[17]:nRaW, void[18]:nRaW, void[19]:aRaW, void[20]:aRnW, CGObject[1]:v, int[5]:v, int[6]:v , void[9]:aRaW, void[10]:aRaW, void[11]:aRaW, int[7]:aRnW,CGObject[2]:aRnW, void[12]:nRaW, void[13]:aRaW, void[14]:aRnW, void[15]:aRnW, void[16]:aRnW, int[3]:v, int[4]:v, char[2]:v, void[6]:nRaW, void[7]:nRaW, void[8]:nRaW, char[3]:aRnW})

Concept X_5 = ({CGObject, CGText, CGBox, CGPoint}, {Point^3:v, char^1:v, void^3:, void^4:, void^5:})

Concept X_6 = ({CGText}, {String^2:v, void^{25}:aRaW, void^{26}:aRnW, Point^3:v, char^1:v, void^3:, void^4:, void^5:})

Concept X_7 = ({CGBox}, {Point^4:v, void^{23}:aRaW, void^{24}:aRnW, Point^3:v, char^1:v, void^3:, void^4:, void^5:})

Concept X_8 = ({CGPoint}, {void^{21}:aRaW, void^{22}:aRnW, Point^3:v, char^1:v, void^3:, void^4:, void^5:})

Concept X_9 = ({Point}, {int^1:aRnW, int^2:aRnW, void^1:nRaW, void^2:nRaW, Point^1:aRnW, Point^2:aRnW, String^1:aRnW})

Chapter 4 A Prototype System Based on FCA

4.1 Introduction

A prototype of concept-based retrieval system (CBRS) is implemented using Java and CORBA, and one model class library and two small-scale class libraries are constructed to test the effectiveness of the system. This is a completely distributed system, and is enterprise Intranet ready and also the Internet ready if appropriate security measures are taken.

The reasons of choosing CORBA as client/server computing model and implementing in Java language are based on following considerations: portability, language independence, and local/remote transparency. As we all know that CORBA is a cross-platform technology, and Java is a multi-platform solution. CORBA thrives in a heterogeneous environment, and Java aspires to be in a heterogeneous one. Java solves the portability problem through the application of Java Virtual Machine (JVM) implementations, and CORBA is adept at solving problems of integration. One of the many benefits of CORBA is that it lets our objects communicate across languages and operating systems. A Java client is able to talk to a C++ object server, and a C++ client is also able to chat with a Java server object. Another wonderful feature of CORBA is local/remote transparency. We write the client and server programs once and can run them either locally or remotely. The clients can invoke objects on the same machine or across the intergalactic network. We don't have to rewrite a line of code. In our case, VisiBroker OSAgent will find our objects where they are. It's part of the ORB (object request broker) magic. With CORBA ORBs clients do not connect to directly with servers as they do with an RPC (Remote Procedure Call) or socket communication. Instead all requests are mediated by the ORB which is the broker that redirects the request to the appropriate server. This server can be

in the same process, across processes, or across network. The ORB makes it all transparent to the client.

4.2 Analysis and Design of the CBRS

4.2.1 Some Aspects of CBRS Prototype System Design

A thorough and robust analysis is a prerequisite for any software project. Then CBRS is no exception. CORBA does not minimize the system analysis and design. An exhaustive analysis of the software retrieval domain is beyond the scope of this thesis, but we will try to focus on the design part of this particular system.

Although CORBA does not impose any new constraints on the analysis phase of the software development, it does limit our design. It must be pointed out that CORBA does not specify things as reliability, fault tolerance, or any other such quality-of-service criteria. These essentially up to the ORB vendors. **Visibroker for Java 3.4** from Inprise Corporation, used in our retrieval system, does provide tools that help achieve these goals (for more details see the manuals of **Visibroker for Java 3.4**). So at least we can expect a reasonable level of fault tolerance if we move our CBRS to the Internet/Intranet. Nevertheless, there is no excuse for shoddy design.

The CBRS has a number of requirements and constraints. Constraint analysis is again beyond the scope of this thesis. We will, nevertheless, mention here what we mean by constraints. These are system attributes, something that is often confused with the system functionality. By that we mean, the user friendliness of GUI, system response time, and so forth. These are worthy goals in themselves, but they don't have anything to do with the CBRS's goals. So in this section, we will ignore all such constraints and describe the CBRS system only in term of what is required for interoperability and effectively delivering the prototype within a reasonable time frame for this thesis. Therefore we have

to leave out some system functionality or relax non-functional constraints such as response speed and reliability. Prototyping is simply to demonstrate what is possible.

A number of requirements are considered during the system design. The design of the CBRS mainly focuses on the following aspects:

- The CBRS should be easy to use, and a user-friendly GUIs of both client and server should be provided to the user and system administrator. The client side GUI should provide a way for the users to easily formulate queries the users are interested in.
- The CBRS should be available through both the Internet and enterprise Intranet, particularly Intranet, which provides the user to interact with the remote retrieval system or even remote class library. In other words, the CBRS should be a distributed system that can work both at same machine and across different networked computers.
- Since the retrieved classes are ready to reuse, the class library should provide an utility to view the source code of the classes if there is such a need to subclass them.
- The CBRS system should provide a tool to browse the classes in the class library in case the users are not familiar or for general browsing.
- Since the CBRS system is a concept-based one, the server part of system should provide a tool to construct a class library, to organize it into a single-valued context, and to calculate all the concepts of the lattice structure based on the context.
- The CBRS system should support both exact and approximate retrievals of candidate classes from the class library.
- The CBRS system should be multithreaded, which allows several users to connect to the server concurrently.

4.2.2 The Distributed CORBA Client/Server Applications

A distributed application is an application whose processing is distributed across multiple networked computers. Distributed applications are able to concurrently serve multiple users and, depending on their design, make more optimal use of processing resources. In

our prototype of CBRS system, a multi-threaded server will be implemented to achieve this goal perfectly. However, in this prototype system, concurrency is not important. Our main purpose is to implement such a system to demonstrate that our methodology is feasible and effective in area of component retrievals for software reuse.

Distributed applications are typically implemented as client/server systems that are organized according to the user interface, information processing, and information storage layers (three-tier model), as shown in Figure 4.1.

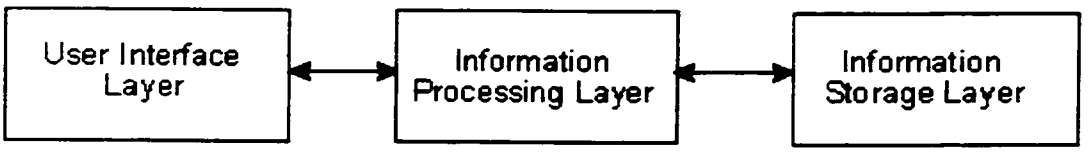

Figure 4.1 The three layer architecture of a distributed system

The applets that run with a web browser (usually Netscape Navigator) are our user-interface layer of components of the distributed CBRS applications. The information processing layer can be implemented as a client, an application or an application support server. In the CBRS system, our retrieve engine is both information processor and storage holder (i.e., the prototype of CBRS is implemented as a two-tier client/server model). Because of security reason, we also need a web server that provided by **Visibroker for Java 3.4** (called **gatekeeper**) to serve file reading need of our client applets. The client/server system deployed on the Internet and the Intranet is schematically shown in Figure 4.2 below.

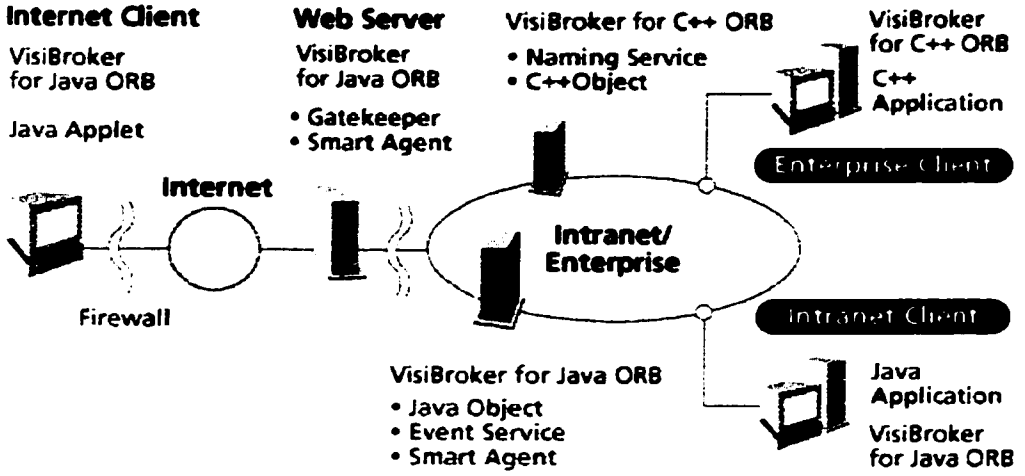

Figure 4.2 The CBRS Client/Server system deployed on the Internet and Intranet

4.3 GUIs of the CBRS System

4.3.1 The Big Picture

CORBA makes use of objects that are accessible via Object Request Broker (ORB). ORBs are used to connect objects to one another across a network. An object on one computer (client object) invokes the remote methods of an object on another computer (server object, also called servant in CORBA 3.0 term) via an ORB.

The core of the CORBA architecture is the ORB that acts as the object bus over which objects transparently interact with other objects located locally or remotely. A CORBA object is represented to the outside world by an interface with a set of methods (operations in CORBA term, they are written in Interface Definition Language --- IDL). A particular instance of an object is identified by an object reference. The client of a CORBA object acquires its object reference (there are many ways for a client to obtain object reference, however, in the prototype of our CBRS, we use URL Naming Service of **Visibroker for Java 3.4**) and uses it as a handle to make remote method calls, as if the object is located in the client's address space. The ORB is responsible for all the mechanisms required to find the object's implementation, prepare it to receive the request, communicate the request to it, and carry the reply (if any) back to the client. The object

implementation interacts with the ORB through either an Object Adapter (OA) or through the ORB interface. Underneath the ORB core is the Internet-Inter ORB Protocol (IIOP) defined on TCP/IP wire protocol. The IIOP specifies how GIOP (General Inter-ORB Protocol, which specifies a set of message format and common data representations for communication between ORBs) messages are exchanged over a TCP/IP network. Figure 4.3 summarizes this process.

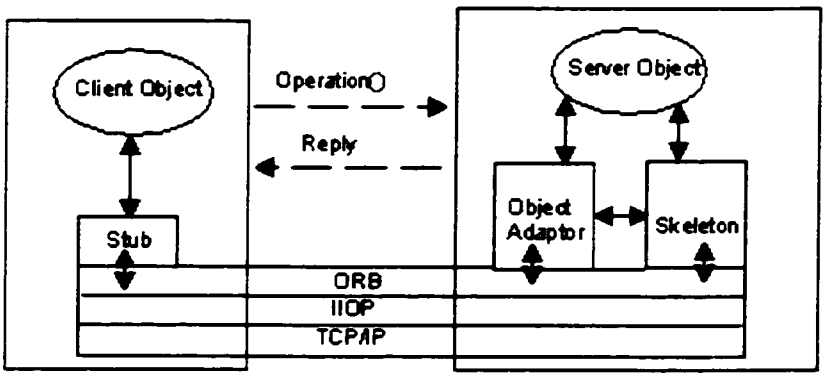

Figure 4.3 How CORBA Client/Server Communicate

4.3.2 The Server Side

The server side has many components that interact each other to provide all the services requested by the client side. These components reside in a package called **Server** (or module in CORBA term). The following is a list of these components with explanations of their functionality.

- **MainServerWindow** is a graphic interface of the server side. It controls the process of hosting server (**mainServer** in prototype of the CBRS) such as starting the server, stopping the server, and also allowing you to start another process to build context from a class library and constructing corresponding concept lattice. This GUI has one information-display-area and three buttons: "Start", "Stop", "Building Concept Lattice". The first two buttons start and stop the hosting server, respectively. Information will be

displayed as text to indicate date of the server start (stop), status, identification and persistence of object implementation. The last button will start another program called **ConImp** to build concept lattice based on the input context that represents a class library. The information-display-area displays the date and time of server start (stop), and server object ID and name as well as status of server object such as persistent or transient. A screen capture of the **MainServerWindow** is shown in Figure 4.4.

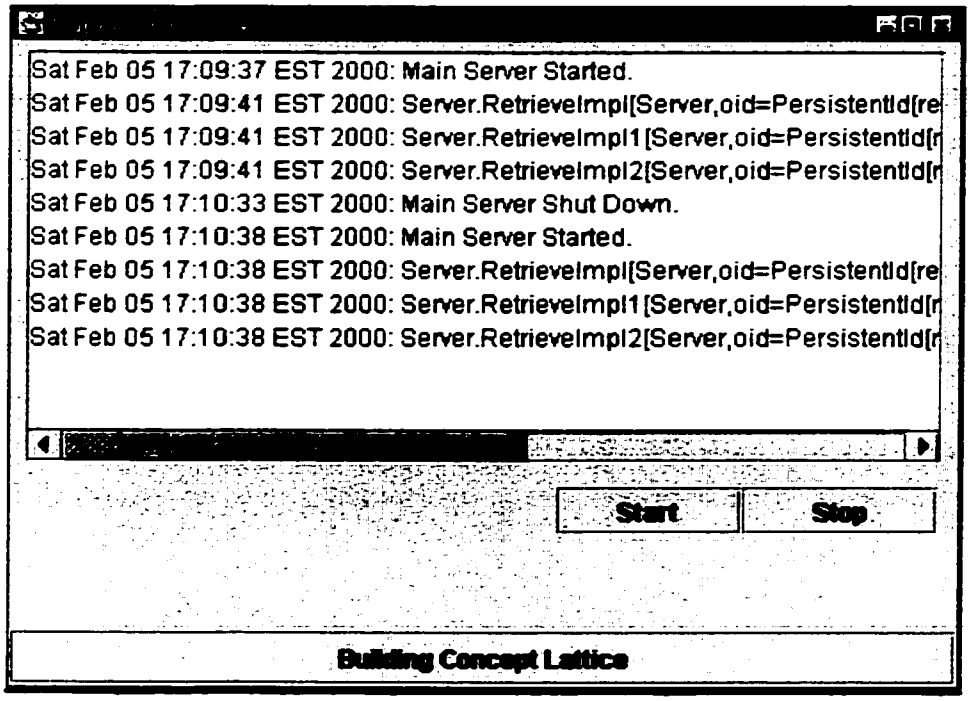

Figure 4.4 GUI of the **MainServerWindow**

- **mainServer** is a thread that implemented **Runnable** interface of Java. It ensures the functionality of multithreading. It creates an instance of retrieval engine's object implementation (called **RetrieveImpl**), and then uses this instance through **URL Naming Service** of **Visibroker for Java 3.4** to create an interoperable object reference (IOR). The client will obtain a handle of **RetrieveImpl** by specifying the URL as a string instead of the object's name through the **URL Naming Service**, too.
- **RetrieveImpl** is an object implementation (server object) of the retrieve engine of the CBRS system. It is the core class of the server side application. It accepts the

information passed by the client side, calls the appropriate functions to perform related operations, and sends the results back to the client side through the ORB. All it's method definitions are defined through **Server.idl** interface.

- **Server.idl** is an interface definition of the object implementation (server object) of the retrieve engine of the CBRS system. It defines the operations that the client can call on the server object through object references.
- **ConImp** is a program that is mainly used by system administrator. It is a console application that accepts input of a set of objects, a set of attributes to organize a context. It also performs operations of calculating each concept of the lattice structure and its label consisting of an object set and an attribute set, as well as the concept's super- and sub- concepts. Its main menu of screen capture is shown in Figure 4.5.

Figure 4.5 Interface of ConImp

- **ClassInfo** is a class that accepts a class name and examines the class, and then returns its class descriptor, superclass, fields and methods related information.
- **Gatekeeper** acts as a web server for our CBRS client/server applications. Since our applets must be run within a web browser like Netscape Navigator, so we must overcome the security restrictions on Java applets, imposed by the browser. Web

browsers impose two types of security restrictions on Java applets (also called Java sandbox security):

(1) They allow applets to only connect back to the host from which the applet was downloaded.

(2) They allow applets to only accept incoming connections from the host from which the applet was downloaded.

The Gatekeeper provides a way to work with these restrictions. In the first restriction, for any server object that is not running on the applet host, the ORB in the applet will try to communicate with the Gatekeeper. The Gatekeeper then attempts to forward any calls from the applet to the server object. This process occurs only for applets configured to talk with the Gatekeeper.

The second restriction is also handled by the Gatekeeper. When the applet creates callback objects, the server object cannot bind to the applet due to the security restrictions. The ORB sets up a special connection between the applet and the Gatekeeper. The Gatekeeper uses the special connection to forward the callback from the server object to the applet (for more details see the **Gatekeeper Guide of Visibroker for Java 3.4**). Its screen capture is shown in Figure 4.6.

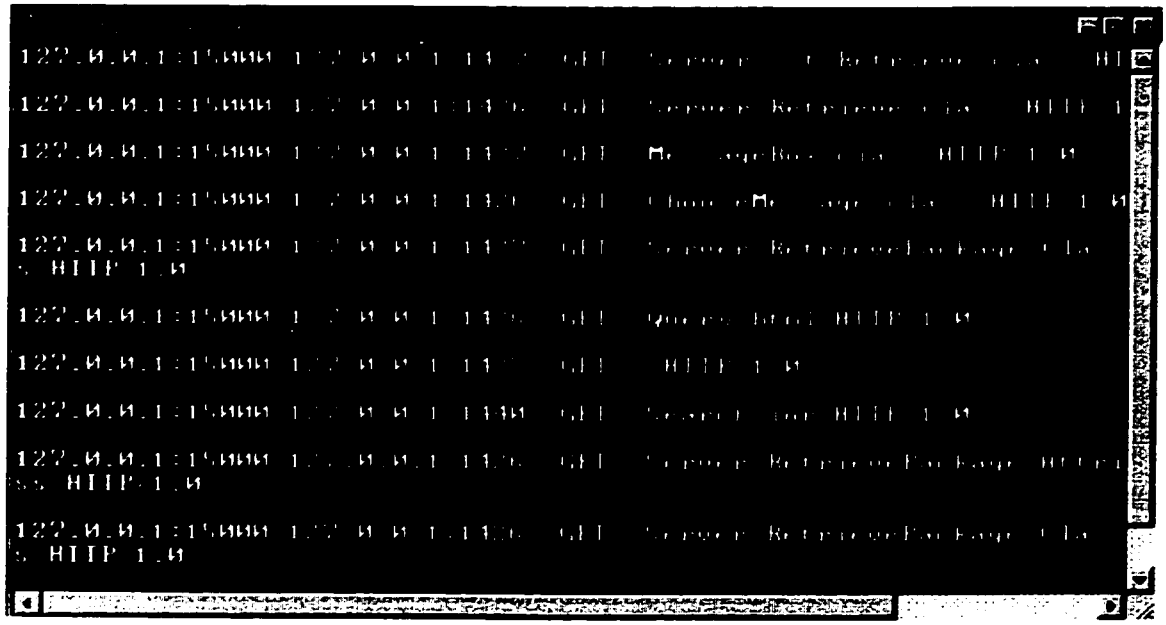

Figure 4.6 Screen capture of Gatekeeper

4.3.3 The Client Side

The client side consists of many applets. These applets have user-friendly GUIs for the users to navigate on the concept lattice, reset the CBRS system, display its subconcepts and all the available composite types, formulate user query, submit the query to the server, view retrieved candidate classes returned by the server and so on. The following is a list of these components with explanations of their functionality.

- **MainInterface** is an applet that controls the whole user-related process, connects to other interfaces for different operations. It displays a list of the class libraries for the user to select, has four buttons: "Browse Class Library" – will lead the user to the **Browse** interface; "Concept-based Retrieval" – will take the user to the **Retrieve** interface; "Save Class Library" – will allow the user to save the class library into a file; "Exit Retrieval System" – will lead the user to **Exit** interface. This applet is executed within a web browser (for example, Netscape Navigator). A JavaScript will run within the browser's status bar to give new user a very brief introduction of what CBRS is and its features. Its screen capture is shown in Figure 4.7.

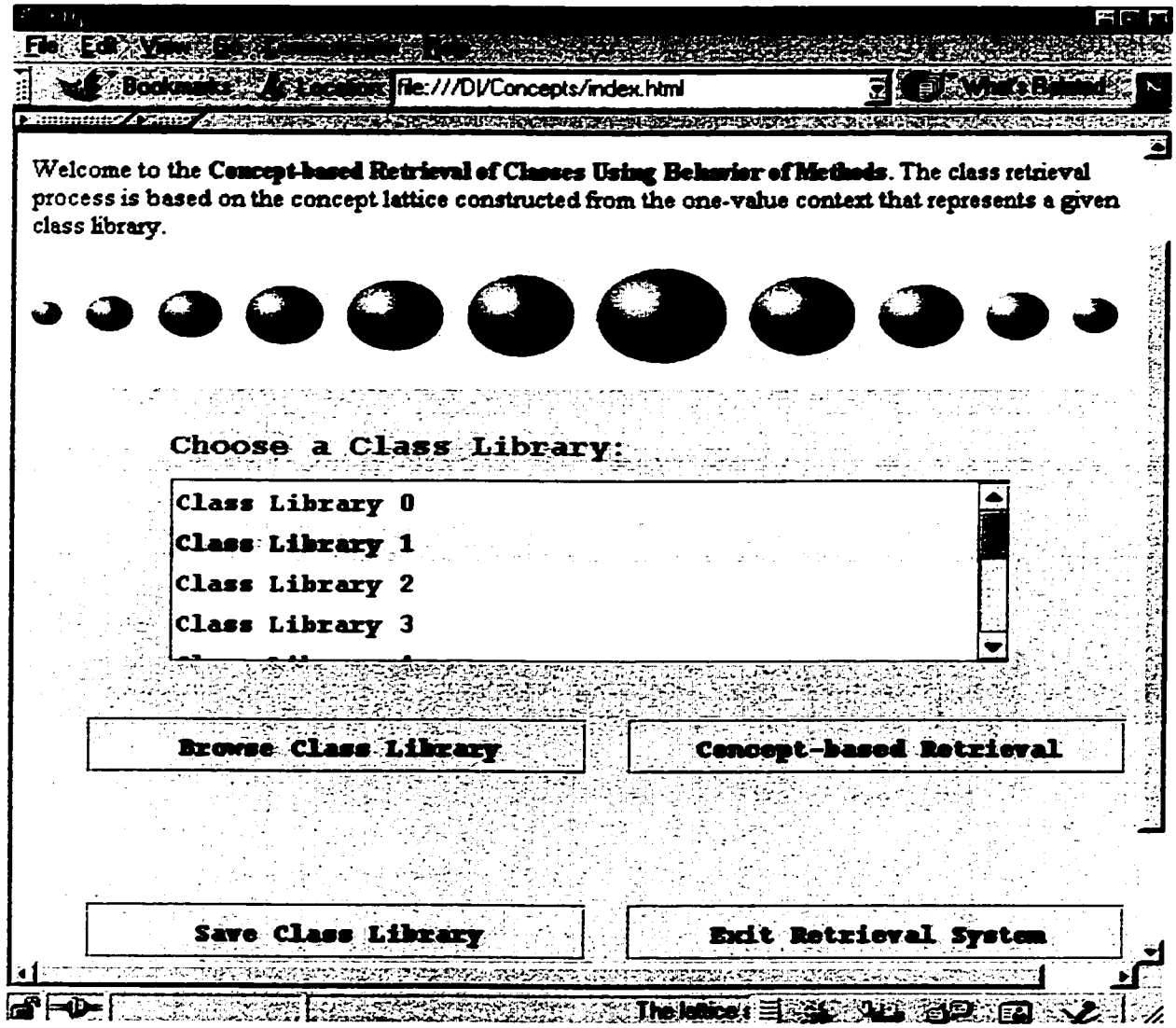

Figure 4.7 The main interface of the prototype of the CBRS system

- **Browse** interface will give the user a chance to look around in a class library. Class library browsing is achieved by navigation of the corresponding concept lattice through another applet called **ConceptLattice**. It can also be used for class library understanding such as relationships between classes.

- **ConceptLattice** is an applet implemented in Java/Swing **JTree** data structure which can be used to display hierarchical data. This applet has three components: a text area that is used to display each node's (concept's) label consisting of class(es) and composite types of all the available variables/methods at this particular node; a graphic

representation of concept lattice as a line diagram; and an expandable tree-like representation of concept lattice, in which if there is graphic icon next to the node, then this node has child nodes. Otherwise, it is a leaf node that linking to the lattice bottom (bottom concept). Clicking the icon will expand the node, and its child nodes will be visible, clicking a node will display the concept's labeling in the text area of the bottom window. The screen dump of this applet is shown in Figure 4.8.

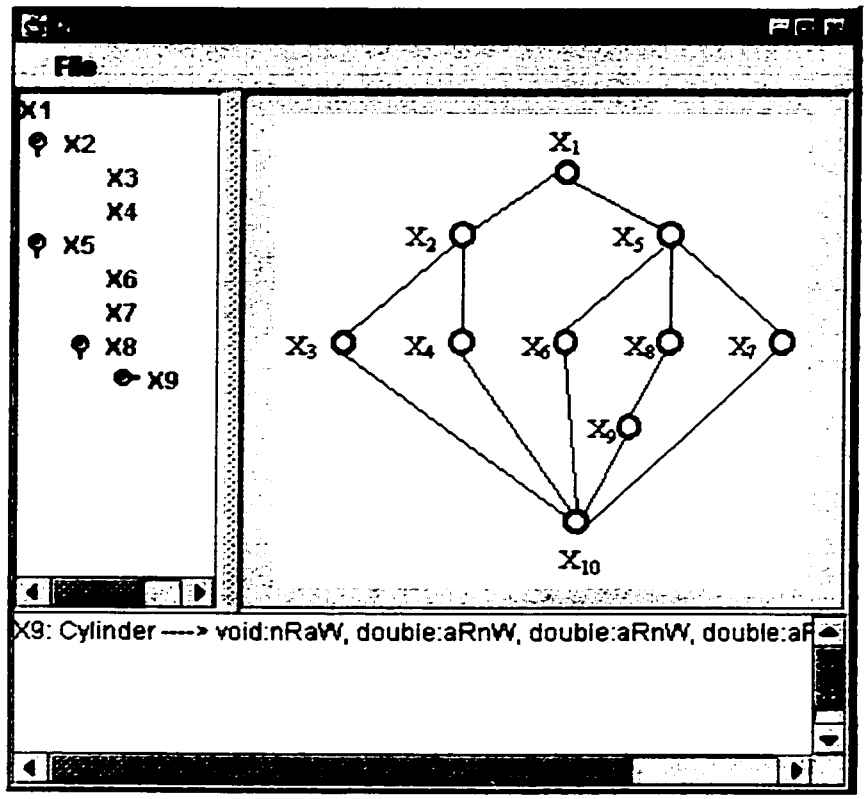

Figure 4.8 GUI of navigating (browsing) a concept lattice

- **Exit** interface will allow the user to exit the CRBS system.
- **Retrieve** interface is most import interface for the client (user). It provides a way for the user to formulate a query through an applet called **Query**.
- **Query** is the core applet for the client side. It has most complicated yet user-friendly graphic interface to the client, has a lot of functionality. There are many components inside this applet which can be run within a web browser like Netscape Navigator. On the left side, the top is a pulldown menu (JCombobox) that allows the user to choose a

concept when the user clicks on the dark solid triangle. When the user chosen a concept, its direct subconcepts will be displayed as a list which is the middle position component on the left side. Below the list is a button called "Back to Main" which will take the user to the **MainInterface** discussed above (see Figure 4.7). On the right side there are two panels: the left panel presents a list of all the available composite types of the concept chosen by the user previously; the right panel displays the composite types selected by double clicking on the items in the left panel. In other words, the right panel holds the user's query. If the user decides to modify the query before submitting the query, he/she can highlight a composite type, by clicking it, that the user want to remove, and then click button "Delete Item". The button "Retrieve Classes" is used to submit the user query to the retrieve engine of the server side application. The result and feedback from the server side will be displayed as a list of the retrieved class in a text area of the bottom of the window. The whole applet's screen capture is shown in Figure 4.9 to illustrate the interface.

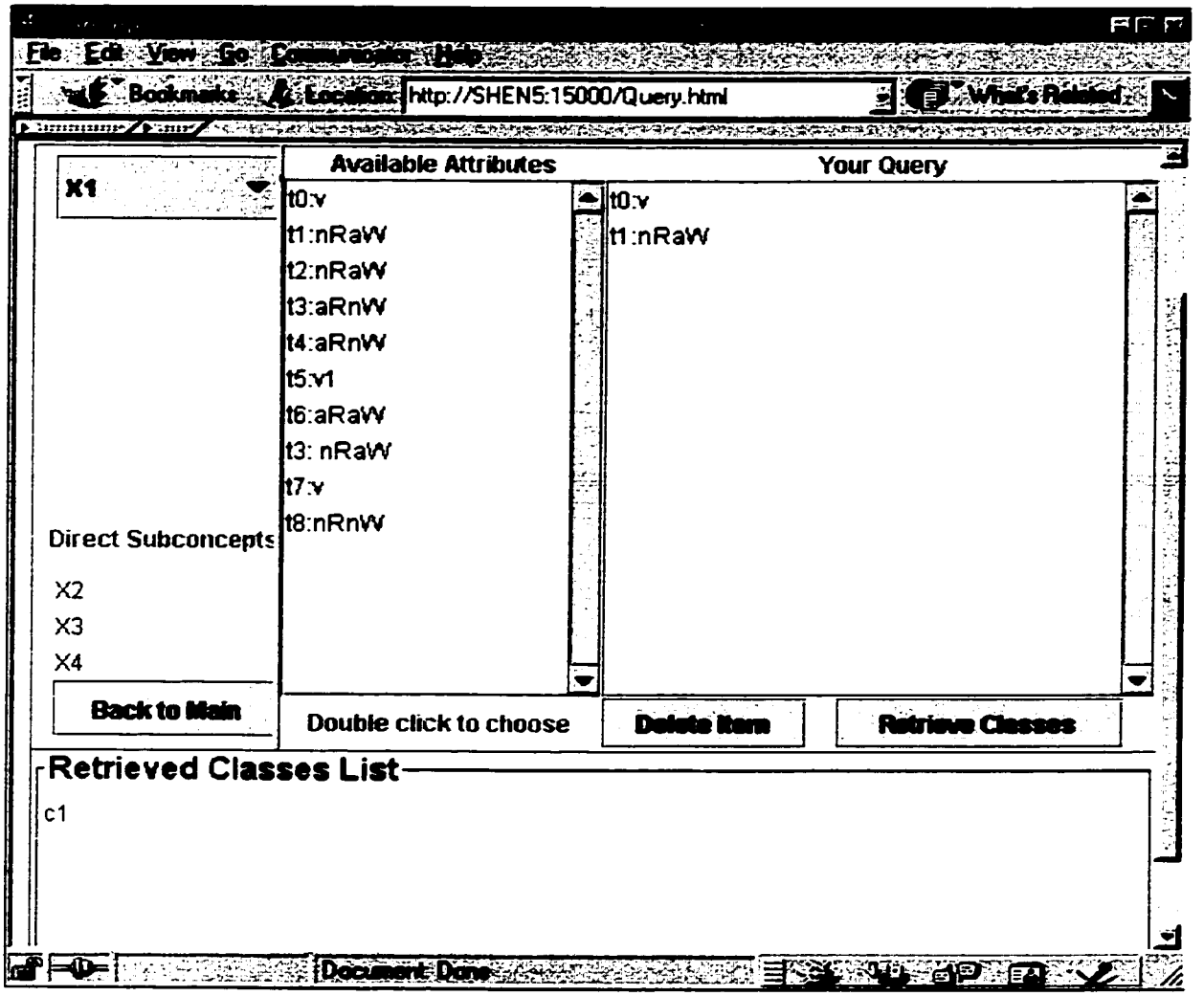

Figure 4.9 The query formulation and retrieval interface

4.4 Building Concept Lattice

In preparing classes for reuse, there are three main activities: choosing classes with high reuse potential, describing these classes, and cataloging/retrieving the classes in a class library. We basically following this procedure for constructing, organizing and building concept lattice of the class library.

4.4.1 Construction of Class Libraries

Two sample and one model class libraries are provided for test the CBRS system. Each class library is constructed as a subfolder within ClassLib folder. The class libraries are organized based on the following considerations:

- Each class is a standard Java class including abstract classes and interfaces, and can only inherit classes from Java APIs or classes in the same class library, rather than from third-party packages. The purpose is to ensure that if a super class is needed for retrieving, the CBRS can find it.
- There is no restrictions on functionality of the classes we can select. They can perform some mathematical calculations on geometric shapes, processing text, graphic drawing, input/output, networking, and even have their own GUIs.
- The types of the variables/methods of a class are not restricted to primitive types, user defined type (class type or object type) is allowed.

4.4.2 Organization of Context and Calculation of Concepts

Among the programs for the support of the FCA, **ConImp** is the most widely used all over the world (this program is available at the following URL address: *http://www.mathematik.tu-darmstadt.de/ags/ag1/Software/DOS-Programme/*). The concept calculation part of **ConImp** is implemented based on Ganter's batch algorithm that is designed for yielding information of concept ordering. **ConImp** allows input and manipulation of one-valued contexts and implications between attributes and calculations of related data from these inputs. In particular, all the data needed for the drawing of related concept lattice by geometric method are provided in **ConImp**.

A formal context is regarded as a mathematical model of table, which relates classes (objects) and composite types (attributes) in a class library. The entries in the table indicates by a letter "x" that the class (object) has the composite type (attribute). An blank

space (" ") character or a period (".") indicates that the class (object) does not have the composite type (attribute). Once we finish all the required entries in the table, then we get a one-valued context for the class library. Of course, before we can input one-valued context, we need to extract the type plus variable access pattern of each available variable/method in a class. This task is not difficult. If necessary we have to scale it because **ConImp** only accepts one-valued context. **ConImp** can only handle contexts with at most 98 attributes and at most 255 objects. Its main menu interface can be seen in Figure 4.5.

Once we have a ready-to-use, one-valued context, we can calculate the concept list, predecessor and successor lists of each concept, as well as assignment list for the concept (including objects and attributes), and then we can draw the line diagram for the concept lattice.

Concept lattice normally grows exponentially in the number of attributes and objects. The complexity of computing all concepts from a given context is $O(2^n)$ in both time and space, where n is the minimum among number of attributes and number of objects in the class library (Ganter 1986). However, in practice the worst case rarely occurs and a polynomial behavior is usually seen (Lindig 1995). If a class library is modified, for example, deletion/addition of a class from/to the class library, its concept lattice needs to be recalculated. This effort usually takes polynomial time as discussed above.

4.5 Retrieval Process

4.5.1 Formal Definition of a Query and a Result

What is a query? Informally, a query is a set of composite types selecting all classes that have at least those composite types. Its formal definition is given below:

Definition 6 (Query) A set $Q \subseteq T^*$ is a query to a concept lattice $G(C^*, T^*, I^*)$. A class $c \in C^*$ satisfies a query, if and only if $\alpha(c) \supseteq Q$ (or $c' \supseteq Q$) holds. The set of all classes satisfying a query is called *result* and is denoted by $[|Q|] = \{c \mid c \in C^*, \alpha(c) \supseteq Q\}$.

Each composite type from a query is introduced by a concept. The extent of the infimum (meet) of all these concepts constitutes the result of the query: all classes of concepts smaller than the infimum are also part of the result.

Theorem 3 Let $G(C^*, T^*, I^*)$ be a concept lattice and let $Q \subseteq T^*$ be a query, then $[|Q|] = \pi_c(\wedge_{t:p \in Q} \mu(t:p))$.

Proof: $\pi_c(\wedge_{t:p \in Q} \mu(t:p)) = \pi_c(\wedge_{t:p \in Q} (\omega(t:p), \alpha(\omega(t:p)))) = \cap_{t:p \in Q} \omega(t:p) = \cap_{t:p \in Q}\{c \in C^* \mid (c, t:p) \in I^*\} = \{c \in C^* \mid \forall t:p \in Q : (c, t:p) \in I^*\} = \{ c \in C^* \mid \alpha(c) \supseteq Q \} = [|Q|]$.

This theorem says that the result of a user query can be found as follows: extracting the extent of the greatest concept of the selected concepts labeled with all the composite types from the user query. Obviously, the intent of the greatest concept must match with the user query. Once the concept lattice for a class library has been calculated, the retrieve engine will use theorem 3 to calculate efficiently the result for any query formulated by the user. Clearly, theorem 3 forms the foundation of our exact retrieval algorithms discussed in section 3.2.3 of the chapter 3.

4.5.2 Formulation of User Queries and Obtaining of Results

The query formulation interface is already shown in Figure 4.9. The user formulates a query by choosing available composite types (called terms in query language) from a list displayed by the CBRS system. When the system is started, it's default state is at the top

concept of a concept lattice that represented a class library. As we discussed in chapter 3, the available composite types at a concept (node) not only includes the intent of this concept but also includes the intents of all its direct or indirect superconcepts following the path upward. So initially, all the composite types of all the classes in the class library are available for the users to choose when the system is just started. Choosing one or more composite types (representing the desired behavior) available from the list will reduce the total numbers of available composite types. Each chosen composite type refines the query. Repeating the above steps eventually narrows search target to one or more concepts whose intents have all necessary composite types requested by the user. One of the important aspect of this process is that the user does not need to know what are the "right" terms for the query in advance. He/she only needs to choose the suggested terms by the system, which seem the most appropriate to catch the target concept. The target concept is called focus of the system. Obviously, the system focus is represented by the extent of the target concept. This kind of narrowing search scope of the target is a downward movement in the concept lattice. More precisely, this process can be divided into two steps: (1) The user selects an additional composite type. As a consequence of the lattice structure, the system supports this selection by calculating all composite types which actually narrow the focus but do not sweep it entirely. It thus can prevent navigation into dead ends (for example, concept bottom). (2) The system calculates the new focus in the lattice as the meet of the actual focus and the defining concept of selected composite type.

The query formulated by the user is saved in **Vector** class. Once the user satisfies with query, this vector that holds all the information the user needs will be passed to the retrieve engine. The retrieve engine will calculate the target concept by processing the user query using the theorem 3 discussed in the section 4.5.1 above. Once the target is found, its extent will be save as a result. Then the result is passed back immediately to the client.

The **Query** applet is a core class of client side. It only provides the interface to interact with the user, and lacks of power of processing the user query. It needs to pass the user query to the server object, i.e. retrieve engine. Passing the user query by the query applet is done through a remote method call to the server object (retrieve engine). The query applet obtained the server object's reference using **URL Naming Service** provided in **Visibroker for Java 3.4**. The **URL Naming Service** provides a way to locate objects without using the **Smart Agent** or a **CORBA Naming Service**. It enables clients to locate objects provided by any vendor (for more details see **Programmer Guide of Visibroker for Java 3.4**) .

An incremental selection from the list of available composite types guarded by the retrieval system allows exact match and completely eliminates the data mismatch and time-consuming deduction phase in a traditional non-concept-based retrieval schemes, and thus improves retrieval reliability. It also allows more precise information to be retrieved from the library.

We now consider a few queries discussed in section 3.3.5 of the chapter 3. The first query $Q_1 = \{t_0{:}v, t_1{:}nRaW, t_5{:}v, t_6{:}aRaW\}$. The exact retrieval system gives the result of class c_4, which is the extent of the target concept X_4. The screen shot of this query is shown in Figure 4. 10 below.

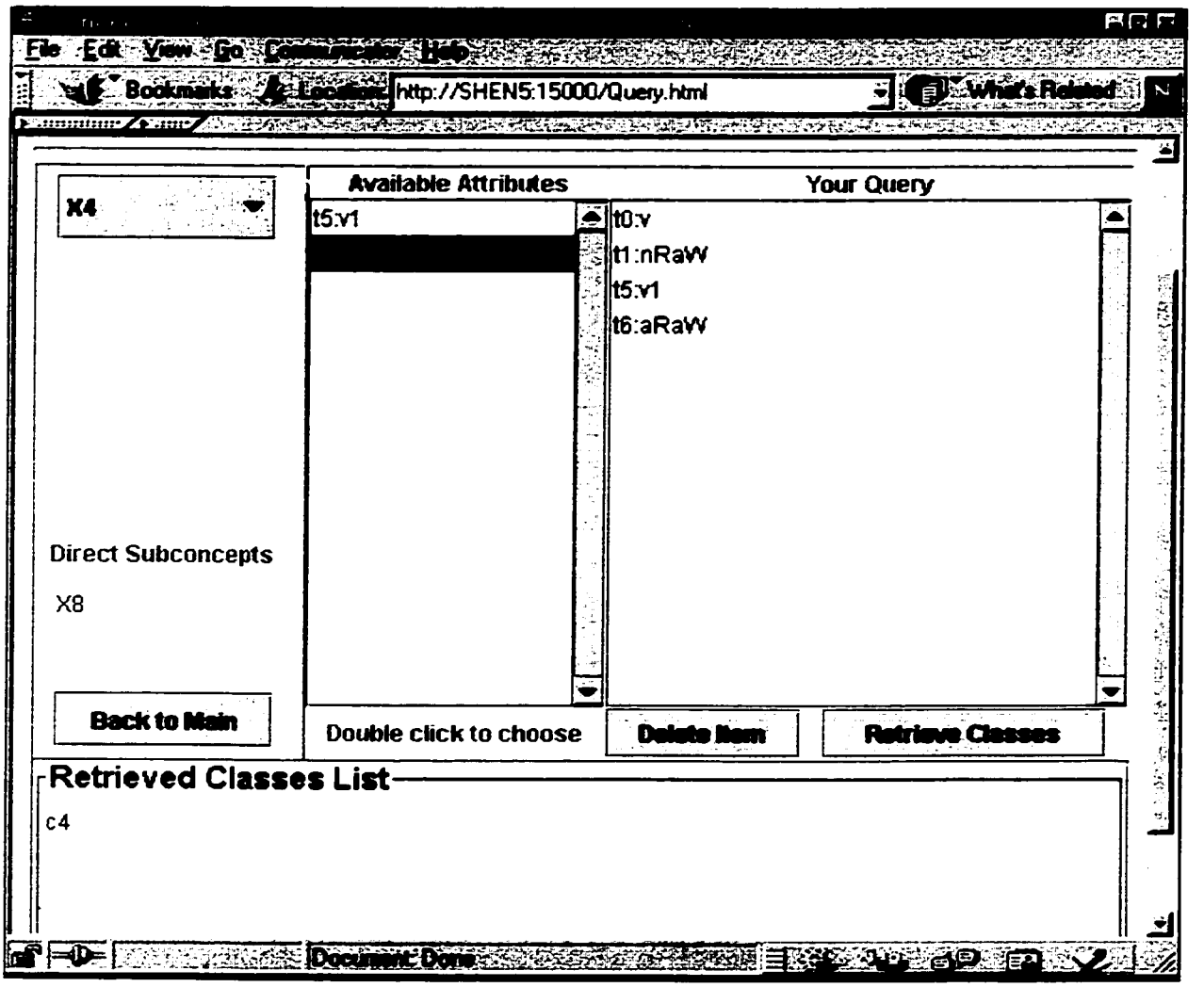

Figure 4.10 Exact retrieval of query Q_1 performed on the model class library

The second query $Q_2 = \{t_0{:}v, t_1{:}nRaW, t_4{:}aRnW, t_8{:}nRnW\}$. The exact retrieval system will display a message ---- "No class(es) matches your query!". For approximate retrieval, the retrieval system will retrieve class c_5 which is extent of the target concept X_5 because the Q_2 is a subset of all the class members of c_5, i.e., the intent of concept X_5, which has six accessible members. In this case $k = 4$ and $l = 2$, the system retrieves the class that has $(k + l)$ members from the target concept. The screen dump is shown in Figure 4.11.

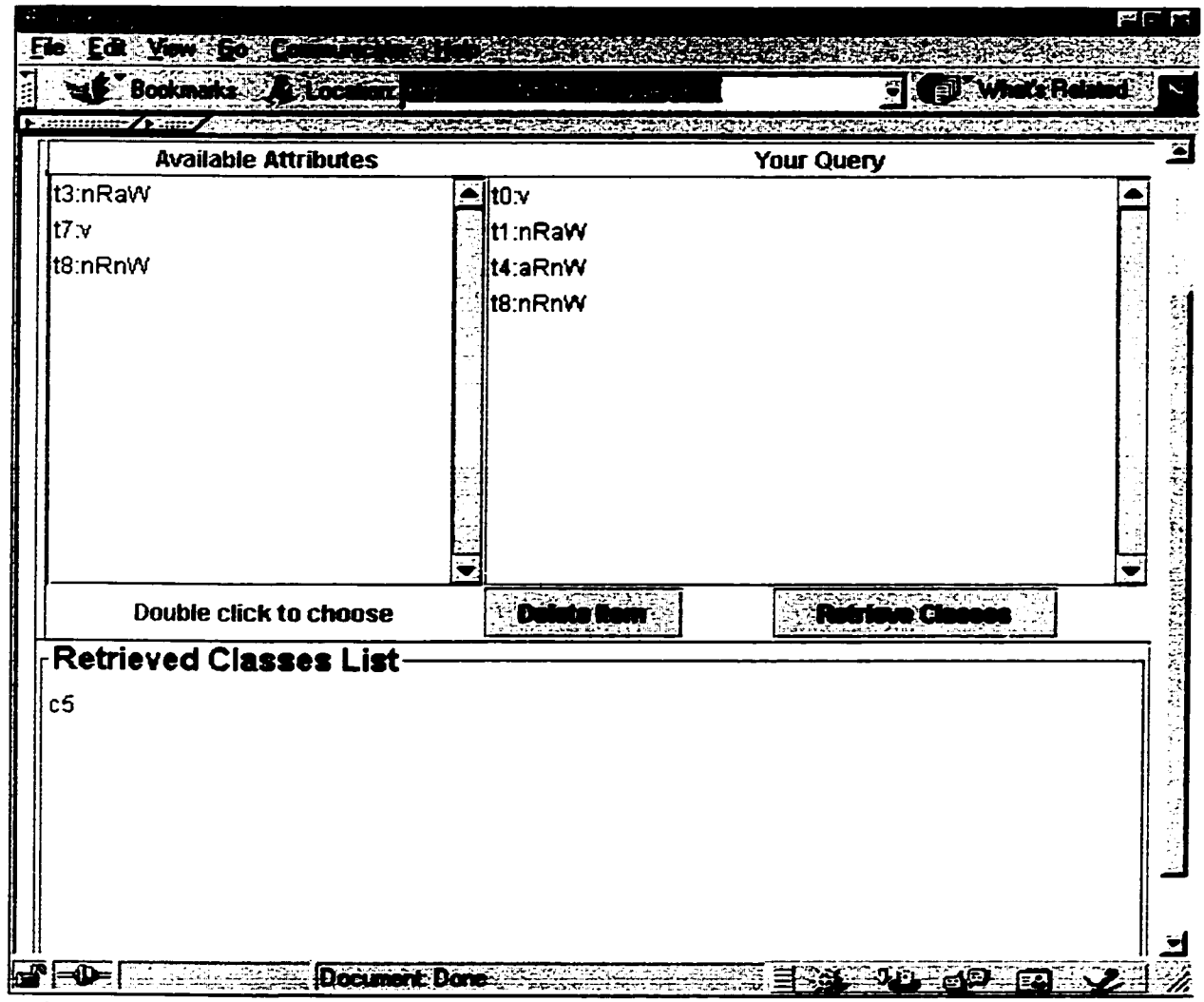

Figure 4.11 Approximate retrieval of the query Q_2 performed on the model class library

The third query Q_3 = {t_0:v, t_1:nRaW, t_2:nRaW, t_3:aRnW, t_5:v, t_6:aRaW}. The exact retrieval system finds the concepts X_1, X_2 and X_4 that are labeled with {t_0:v, t_1:nRaW}, {t_2:nRaW, t_3:aRnW} and {t_5:v, t_6:aRaW}, respectively, and finds the concept bottom as the greatest lower bound among X_1, X_2 and X_4. It then retrieves no class because the extent of the concept bottom X_8 has empty set. The exact retrieval system just displays a message ---- "No class(es) matches your query!" as a feedback to the user. Its screen capture is shown in Figure 4.12.

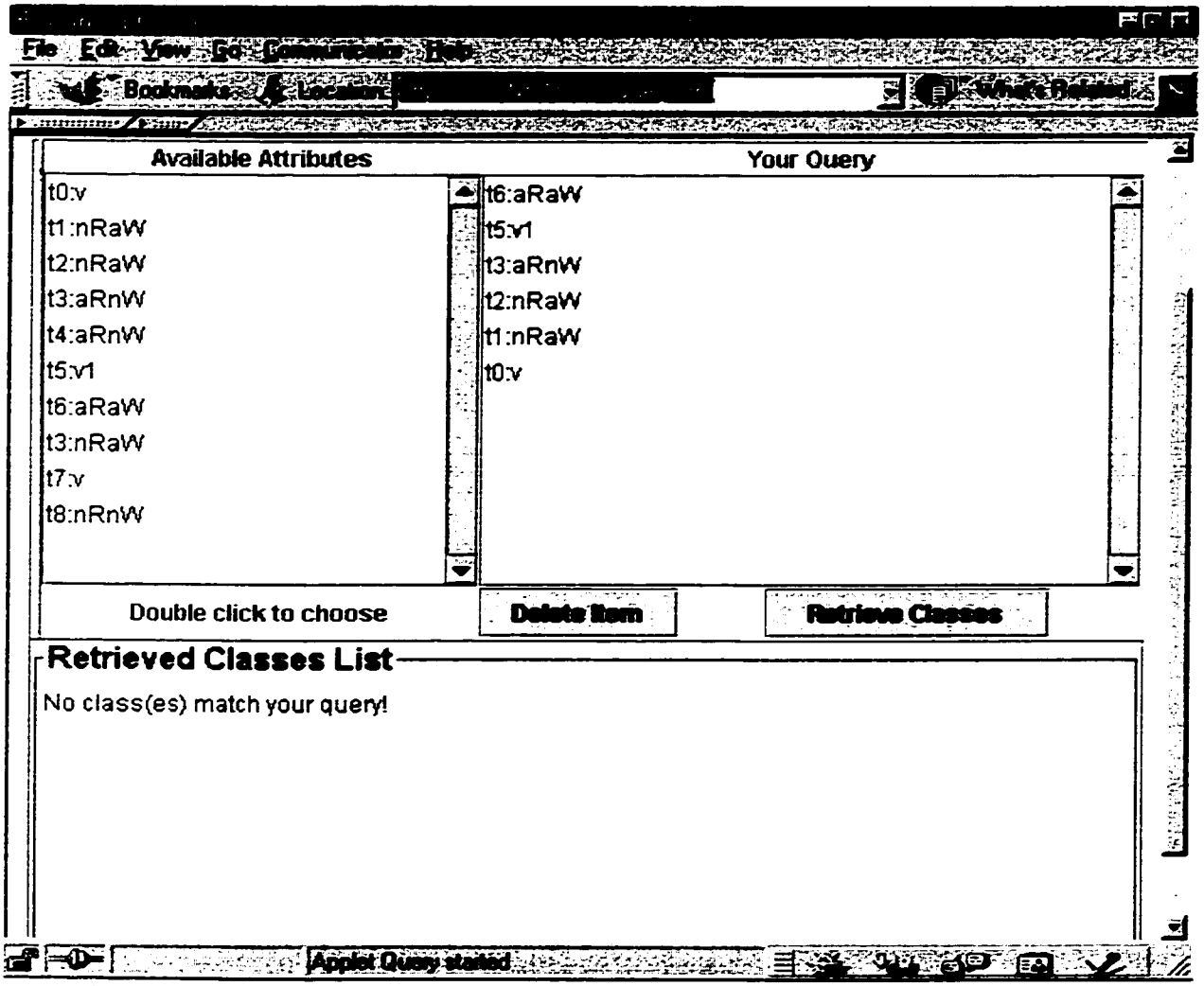

Figure 4.12 Exact retrieval of the query Q_3 performed on the model class library

The fourth query is again Q_2 = {t_0:v, t_1:nRaW, t_4:aRnW, t_8:nRnW}. This time the user wants to find a class that has *l* few members than the desired class, i.e., any class that has first three members from Q_2 (k = 4, l = 1 and (k - l) members in this case). The approximate retrieval system will perform the search using all possible subqueries, i.e. {t_0:v, t_1:nRaW, t_4:aRnW}, {t_0:v, t_1:nRaW, t_8:nRnW}, {t_0:v, t_4:aRnW, t_8:nRnW} and {t_1:nRaW, t_4:aRnW, t_8:nRnW}, and finally retrieve class c_3 because one of the subqueries containing composite types {t_0:v, t_1:nRaW and t_4:aRnW} is a subset of the concept X_3's intent. The screen dump is shown in Figure 4.13.

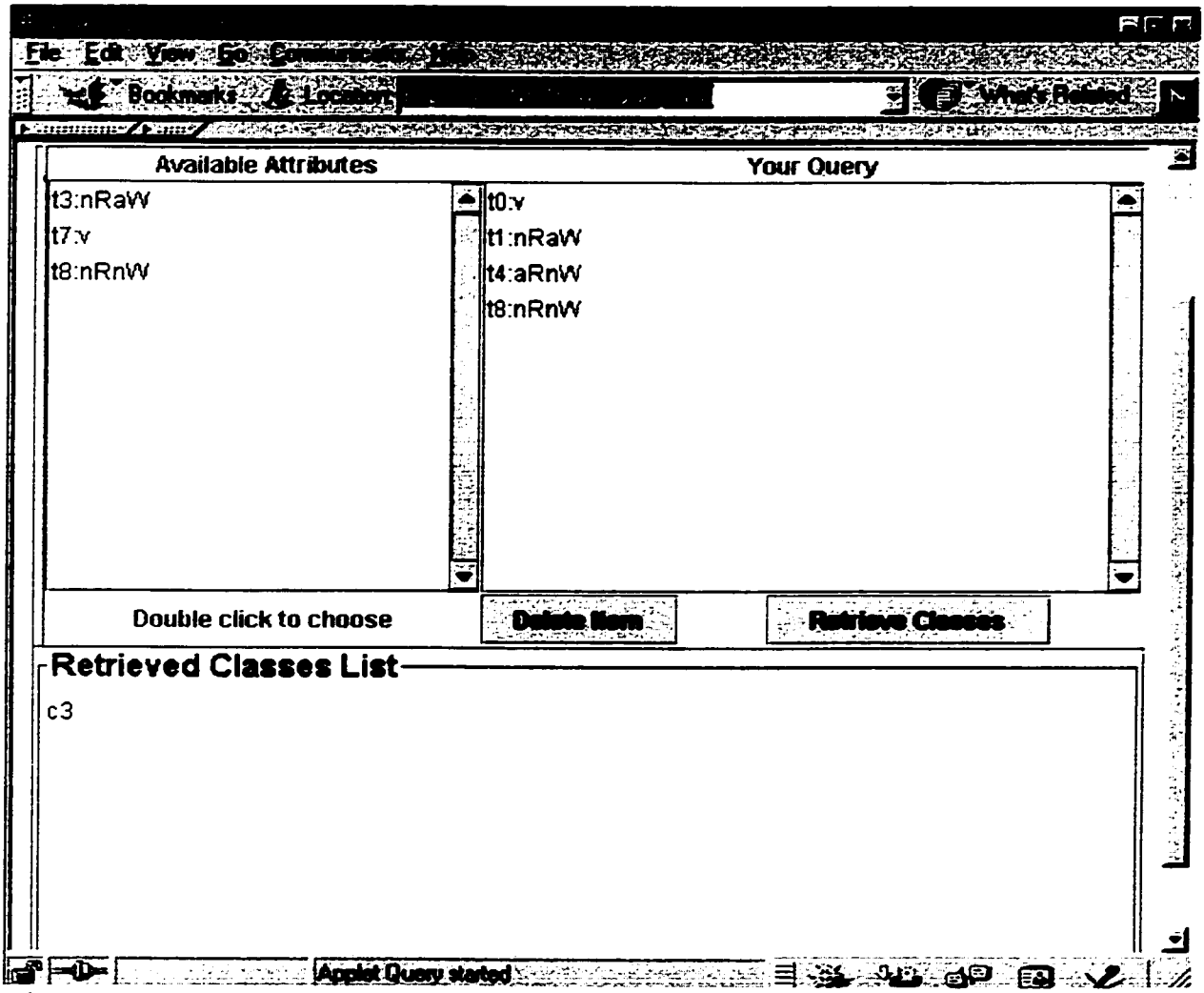

Figure 4.13 Another approximate retrieval of the Q_2 performed on the model class library

As we can see from the above examples the query formulation by the user is incredibly easy. It is well known that large-scale search engines for the WWW usually retrieve documents quite effectively, but they can be considered imprecise because they do not exploit and hence retrieve the semantic contents of Web documents. However, the query constructed by the user in our CBRS has clear semantics, thus the retrieved result is comprehensible to the user. In other words, the concept lattice allows retrieve engine to process queries more efficiently and provides optimal feedback to the user.

4.6 Class Library Browsing - Navigating Concept Lattice Directly

We already said enough about the difference between browsing and retrieving in last chapter of section 3.2.1. The purpose of browsing a class library is two folds: (1) the users are not familiar with the retrieval system and have no concrete knowledge what they are looking for; (2) the users want to go through browsing a class library to achieve the purpose of retrieval of partially-satisfied classes because there is not always an exact match between the user's requirements and available classes in the class library. In the late case, the user wants to look around for a class that is close enough to be adapted to his/her needs. The hierarchy generated from the concept lattice of a class library can be used for browsing and understanding the relationships between the classes. The reason that we need library understanding is that more benefits can be achieved only by retrieving an suitable component, understanding it and adapting it to the new environment.

Navigating directly on a concept lattice hierarchy of a class library actually plays the roles of both browsing and searching the library. The navigation is achieved through an applet called **ConceptLattice** that is implemented in Java/Swing **JTree** data structure. The Swing **JTree** is a perfect choice to display hierarchical data. The screen capture of this applet is already shown in Figure 4.8.

As mentioned early this applet has three components: (1) A text area that is used to display each node's (concept's) label consisting of class(es) and composite types of all the available variables/methods at this particular node. (2) A graphic representation of concept lattice as a line diagram (a gif format image). (3) An expandable tree-like representation of concept lattice; this is a very high-level view of a concept lattice in which only the top concept and its direct subconcepts (child nodes) are displayed at the starting stage of the system, which is very important for a very large of class library

because otherwise the user will be flooded with so many choices that he/she gets lost; to see the whole concept lattice we have to expand every node that is expandable (if there is graphic icon next to the node, then this node is expandable and has child nodes. Otherwise, it is a leaf node that is directly linking to the lattice bottom). Clicking an iconized node will expand the node, and all its child nodes will be visible to the user, and then clicking a node will show the corresponding concept's labeling, in the text area of the bottom window, including information of class and its available composite types. This information is vital to user because the user relies on this information to make the decision that if this is the right choice for reuse purpose.

A wonderful feature of **JTree** representation of a concept lattice constructed from a class library is that the users can visit any node they want without any restrictions. In theory, the **JTree** is a special graph. Therefore the users have two choices of visiting a particular node: depth-first search (DFS) and breath-first search (BFS). In the DFS, the user wants to go as far away from the starting point as quickly as possible. In the BFS, the user likes to stay as close as the starting point, and visits all the nodes adjacent to the starting node, and only goes further afield. Both methods will reach eventually all connected vertices (nodes). A few screen shots of library browsing from the model class library and class library2 are shown in Figures 4.14 - 4. 17.

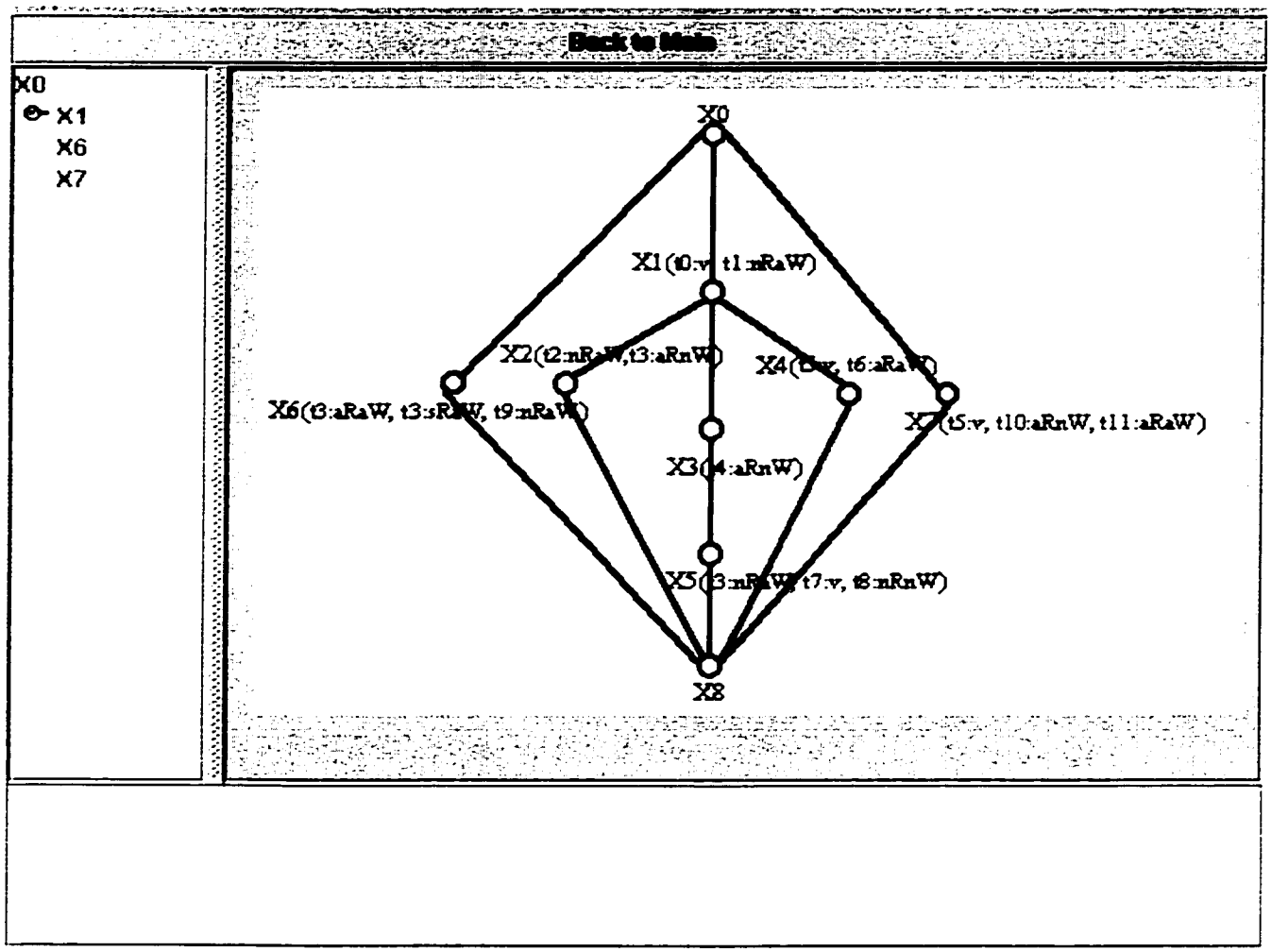

Figure 4.14 Initial screen shot of browsing the model class library

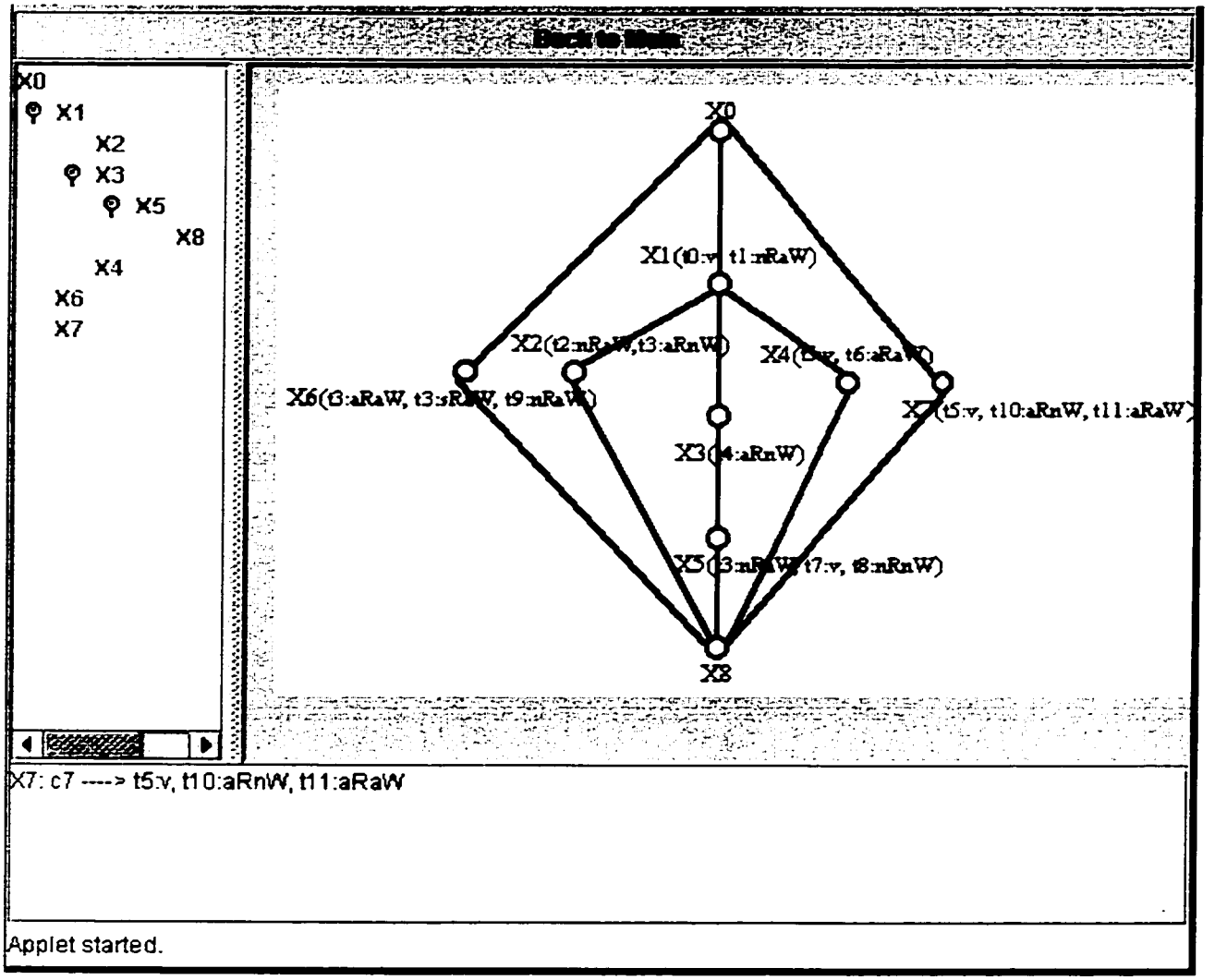

Figure 4.15 The screen shot of browsing the model class library after expanding

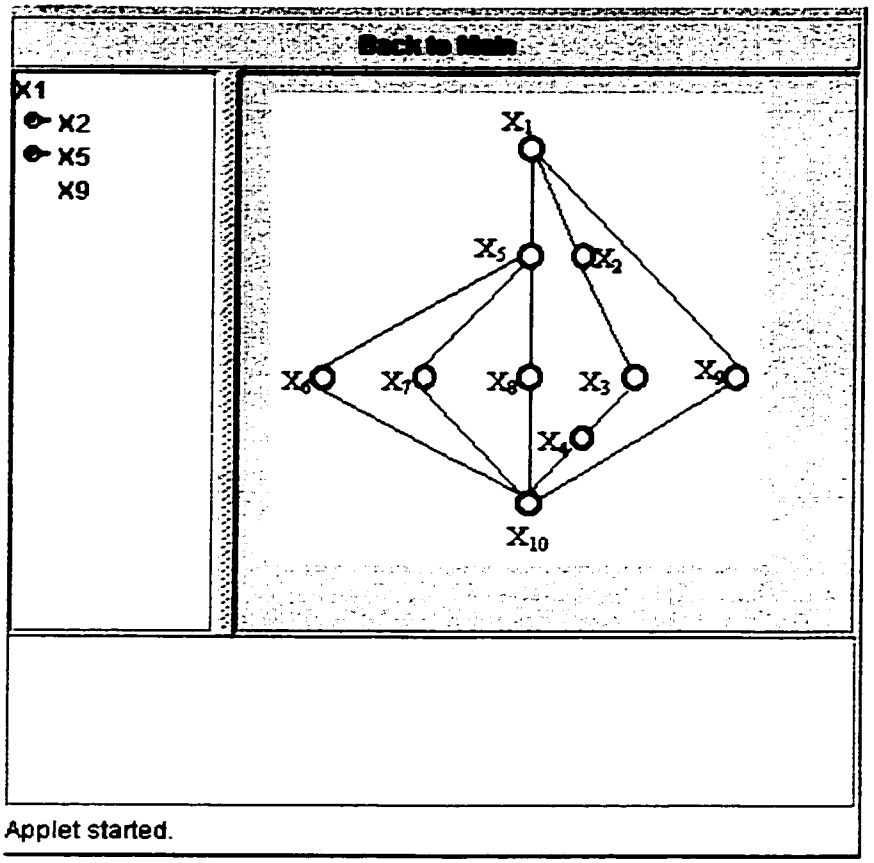

Figure 4.16 Initial screen shot of browsing the class library 2

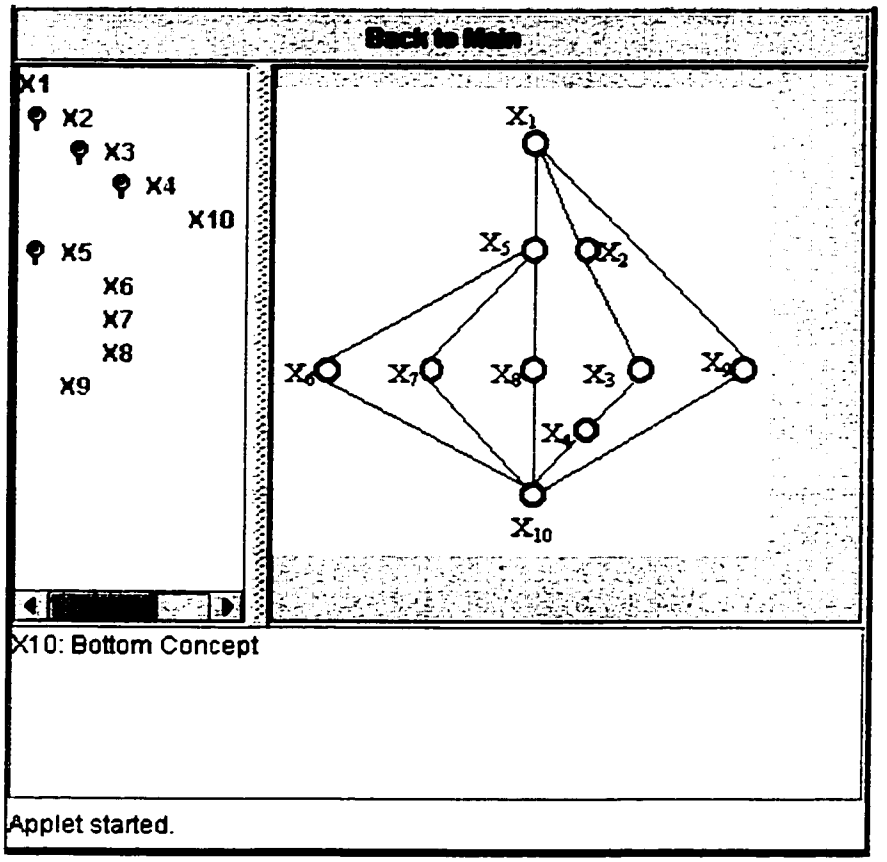

Figure 4.17 The screen shot of browsing the class library 2 after expanding

The advantage of using concept lattice is that we can incorporate more attributes to a class such as a sample behavior, a very good class description for each node (concept) that will also present to the user, which greatly enhances the usefulness of the hierarchy for browsing purpose.

4.7 A Scenario of Concept-Based Retrieval

In this section, we will provide an example to show in details how the CBRS works, and how a client talks to the server and how the client get the result from server.

Consider the class library 2 in chapter 3. In this class library there are total 8 classes. The class members and relationships among all the classes are shown in Figure 3.6 through UML notation. Its corresponding context and concept lattice are shown in Table 3.5 and Figure 3.7, respectively.

When the CBRS system is started, it will display all the 44 available composite types of the classes in this class library due to the fact that the system begins its journey downward from the top concept X_1 in Figure 3.7. For the beginner users, they only see the two columns on the top panel of the Query applet (see Figure 4.18 below): the left column that displays the 44 composite types available for the user to choose --- *Available Attributes*, and the right column that displays the items chosen by the user --- *Your Query*. For advanced users, they can drag the vertical divider bar to the right to see the current status of the system, i.e., the current concept the system stays at and its direct subconcepts. The user even can click on the "top-down triangle" icon within a combo box on the upper left corner to see how many concepts in this library.

Suppose the user is looking for a class that has following 20 members in Q = {int^3:v, int^4:v, char^2:v, void^6:nRaW, void^7:nRaW, void^8:nRaW, char^3:aRnW, CGObject^1:v, int^5:v, int^6:v ,void^9:aRaW, void^{10}:aRaW, void^{11}:aRaW, int^7:aRnW, CGObject^2:aRnW, void^{12}:nRaW, void^{13}:aRaW, void^{14}:aRnW, void^{15}:aRnW, void^{16}:aRnW}. All these items can be found on the left column. If the user double-clicks any item from q = {int^3:v, int^4:v, char^2:v, void^6:nRaW, void^7:nRaW, void^8:nRaW, char^3:aRnW}on the left column, and then this item will be put it into his/her query. They can only put one item into the user query one time. The system will automatically reduce the total numbers of available composite types from 44 to 24, which allow the user to choose on the left column, and then automatically change to the concept X_2. The system will stay at concept X_2 until these 7 items of the q go into the use query. Choosing any item further from the rest of Q (i.e., Q - q) will bring the system to concept X_3 and again reduce the total numbers of available composite types from 24 to 17. We can see that the incremental selection of composite types rapidly narrows down the search space. Once the user fills the query on the right column, then he/she can click "Retrieve Classes" button, and the query Q will be sent to the retrieve engine on the server side. The retrieve engine will calculate the meet of concept X_2 and X_3 based the Q. From the Figure 3.7 we can see that the meet is the X_3. The result (PrintCGrid) is the extent of the concept X_3, which will be passed back to the client by the server. Figure 4.18 shows a complete picture of this scenario. When user

finishes the above query and wants to start another query he/she can simply click on the triangle icon on the upper left corner and then choose the concept X_1, the system will be reset to the default state.

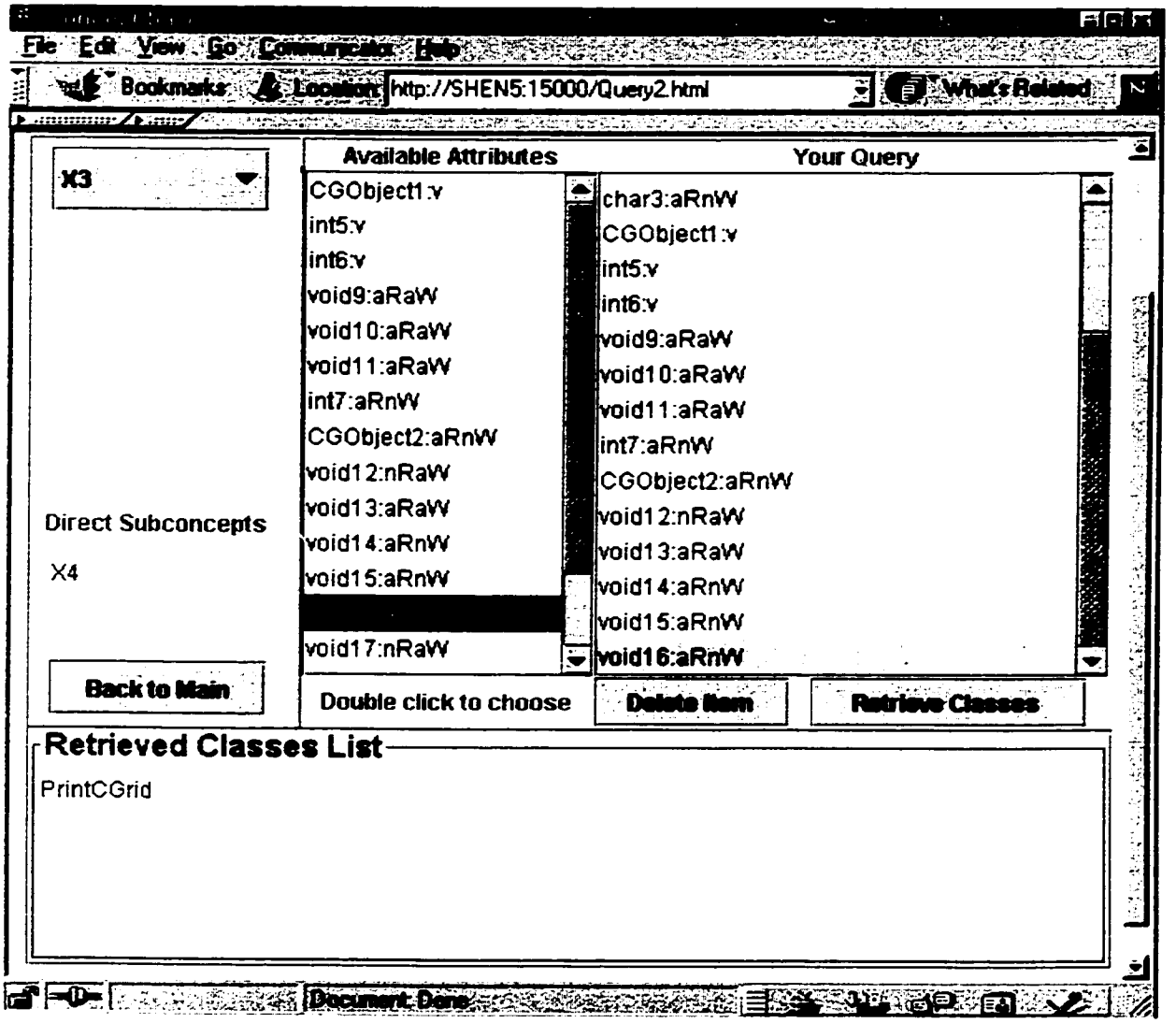

Figure 4.18 Screen capture of retrieving a class from the class library 2

Chapter 5 Evaluation and Conclusion

5.1 Evaluation of the Methodology and Implementation

The value of a reuse library lies in being searched easily, and the fact that suitable component can be effectively and precisely retrieved from the library. The criteria mentioned in Niu Thesis (Niu 1999) is still applicable for evaluation of the methodology proposed in this thesis. In the rest of this part, we do an evaluation of the methodology and implementation based on similar criteria, i.e., general-purpose, scale-up, retrieval precision and recall, and retrieval efficiency. Technically speaking, experiments should be carried out to compare different retrieval techniques and build from scratch. In the following we will try to give a qualitative evaluation of the methodology against the above mentioned criteria.

5.1.1 General Purpose

General purpose means that the technique could be applied to multiple domains or applicable in as many situations as possible.

The methodology is an attributed-based retrieval system which depends on the explicit and incremental choice of functional properties and thus is very suitable for reuse purpose. A very simple addition of object-based retrieval style would make the methodology equally suitable for library understanding and reengineering domain because the object-based retrieval style exposes implicit conceptual similarities of components: the intent of the focus (target concept) contains all properties which are common to all

selected component; its extent also contains all other components which share these properties, even if they have not been selected explicitly (Fischer 1998).

The methodology only needs a class library and a user query to perform retrievals of class components, regardless of OOP languages and development environment/operating systems. For example, the methodology can be applied to a C++ class library where even multiple-inheritance is a frequent phenomenon which has no significant impact on the way of organizing context of the class library except changing some contents of the incidence relations appeared in the context. Successful applications of a FCA-based approach to class design and maintenance of both C++ and Smalltalk class hierarchies (Godin et al 1998 ; Snelting and Tip 1998) also provide supporting evidence of language and domain dependence of the methodology.

The CBRS is implemented using a distributed CORBA client/server architecture and Java/Swing technology. The unique combination of Java/Swing and CORBA means portability, language independence, and local/remote transparency. The implementation is not only Internet ready but also enterprise intranet ready. A C++ client can make full use of the services provided by retrieve engine without any code modification on the server side. Similarly, in order to improve performance, we can re-implement retrieve engine with C++ and still use the Java client with professional GUIs whose appearance (look & feel) are independent of operating systems due to the nature of Java/Swing.

5.1.2 Scale-Up

Concept lattice normally grows exponentially in the number of attributes and objects that appeared in the context. In practice, however, the worst case rarely occurs and a polynomial behavior is usual in both time and space complexity. According to Lindig, the computing time ranged from 0.1 to 1000 seconds for approximate 50 to 8000 concepts (which is far below the upper bound) on a SPARC station ELC (Lindig 1995).

The computation time of the concept lattice for a class library is a preprocessing effort, which can be regarded as an off-line phase and thus separated from class retrievals. The end user only uses the preprocessed and prebuilt lattice structure to perform retrievals. The semantic properties of all classes in the class library are reflected in the intents of all the concepts. Therefore using the above techniques, a small to large libraries that contain up to thousands of classes can be handled by the methodology because the off-line computation time of the concept lattice is not critical to the retrieval process.

5.1.3 Retrieval Precision and Recall

Precision is the number of relevant components retrieved by a user query, over the number of all the components retrieved by this query. Recall is the number of relevant components retrieved by a user query, over the number of all relevant components in the class library. Effectiveness means that the retrieval results should show high precision and recall. Theoretically, maximum effectiveness means that all relevant components should be retrieved (recall = 100%) and result of a user query should contains only relevant components (precision = 100%). A study of typical reuse approaches reported in the literature shows the numbers in the range of 40 -60 % for both values of precision and recall (Biggerstaff and Perlis 1989).

The concept-based approach achieves class retrieval by traveling through the lattice structure by following certain paths. It visits the necessary and relevant nodes only, no more no less, to match the user query. The exact retrieval mechanism guarantees that the number of relevant classes equals to the number of all the classes retrieved by the user query in most cases. Both in theory and in practice, the exact retrieval produces almost 100% precision due to incremental selection of composite types in the concept-based approach. This high precision can never be achieved in other comparable retrieval techniques. Since the approximate retrieval is based on the exact one, it will retrieve all

the classes whose members partially match with the user query. A very possible situation is that the system retrieves all the superclasses or subclasses of a particular class that is an extent of the target concept. The recall is also relatively high. The detailed value of the recall depends on the user query.

5.1.4 Retrieval Efficiency

Retrieval efficiency in particular means that search time must be short.

The CBRS retrieves classes by traveling through the lattice structure by following the paths calculated by the system. It visits the necessary and relevant nodes only, no more no less, to match the user query. In other words, the retrieval system searches only relevant concepts to perform the class retrieval, the rest of concepts is ignored by the system. So the retrieval efficiency is high.

In practice, the implementation requires a lot of resource to perform normally. The CBRS starts slow if all the required resource is not enough. The **Gatekeeper**, which acts as a web server for the system, needs a lot of memory and runs slow. The vendor also needs to improve its performance. Once the system is started and in a state of ready to retrieve, it performs relatively fast. Of course, if the CBRS is deployed either on the Internet or enterprise intranet, the system performance depends on many other factors such as network traffics, the resource of the client computer and so forth.

5.2 Comparison with Other Retrieval Approaches

Part 1.3 of the chapter 1 already reviewed three traditional, major retrieval techniques reported in the literature: external index, internal static index (structure-based scheme)

and internal dynamic index (behavioral-based scheme). Strictly speaking, the methodology proposed in this thesis does not belong to any category of three traditional approaches. However, its surface features seem to belong to both category 2 and category 3 this is because the methodology is designed to take advantages of the two approaches and discard their dross. Niu's methodology (Niu 1999) is of category 2, too. Both similarities and differences between our methodology and the three approaches can be found through comparisons.

5.2.1 Similarities

There are some similarities between our concept-based methodology and the other three retrieval approaches, which can be summarized as follows:

- The concept-based approach provides a way to represent a class as a multiset of types. It adopts similar technique that used also in type-based schemes and execution-based approaches for some C functions.
- The concept-based approach organizes class library into a context (relation) while type-based approach also organizes the class library by grouping classes based on either the number of methods or the types (including return type and argument type) of methods of these classes. So it has very similar surface features with that of type-based and some execution-based schemes.
- The concept-based retrieval approach follows the procedure of construction of user query, sending it to the retrieve engine, and class retrieval by matching query. The spirit of retrieval pattern is very similar to that of both type-based and execution-based as well as classification-based schemes.
- The concept-based approach allows both exact and approximate retrievals. The type-based, execution-based and classification-based schemes also provide similar retrieval styles. This is because query matching can be roughly divided into two categories: exact match and partial match.
- The concept-based approach explicitly uses variable access patterns of methods of a class (component) as the class behavior. For classification-based retrieval, the behavior

of the components is concentrated implicitly into terms used in each facet. The terms are chosen from a vocabulary that is constructed from the specific domain, not from the component itself. For signature-matching (type-based) retrieval, the behavior of the components is very dilute, residing in the type signature of each function of a module. Although the behavior is represented differently in all these three approaches, they all catch the behavior of the components by some kind of indexing (extracting) either from the specific domain or static properties of the components themselves.

5.2.2 Differences

There are significant differences between the three traditional approaches and the concept-based approach. These differences mainly come from the ways of organizing libraries and the mechanisms of query processing used by retrieve engine. They can be summarized as follows:

- The concept-based approach organizes a class library into a context (relation) and then further builds a corresponding concept structure which has strong mathematical properties and can be processed and analyzed by very precise algebraic means. The structure is a very rigid lattice, has very strict ordering of its nodes and has strong formal and mathematical foundations. In contrast, the library structures of both type-based and execution-based schemes are very loose one, at best a unbalanced tree-like structure maybe obtained. In most cases, only some degree of arrangements of components is achieved through grouping and linking. Most parts of the library structures, like the one used in classification-based scheme are amorphous and random-ordered or have only loose linking and grouping. The library structure lacks of formal or mathematical foundation.
- The concept-based approach extracts class behavior from static properties of class components. In contrast, execution-based scheme captures the component behavior by executing test data on the methods of the class component; classification-based scheme captures the component behavior by extracting terms from a specific domain not from the component themselves.

- The concept-based approach achieves class retrieval by traveling through the lattice structure by following certain paths. It visits the necessary and relevant nodes only to match the user query. So retrieval efficiency should be high. The retrieval precision is also very high because of an incremental query construction and dynamic heading of the system focus. The class retrieval of the execution-based approach heavily relies on execution of every class in the library. The process of matching and selecting components for retrieval are very expensive because the behavior of each component is dynamically calculated using the program part of the query.

5.3 Advantages of Using Concept-Based Retrieval System

There are many advantages of the CBRS if we compare our concept-based retrieval system with other major retrieval approaches. Major advantages are summarized as follows:

- Efficient organization of classes indexed with type plus variable access behavior in the reuse library saves disk space. The resulting concept lattice can be seen as a hierarchical clustering of the classes. Classes in a class library are partitioned into sets, and the lattice structure imposes a taxonomy on these class sets. The lattice structure improves the retrieval efficiency by searching only relevant concept space.
- An incremental selection from a list of available composite types guarded by the retrieval system allows exact match and completely eliminates the data mismatch and time-consuming deduction phase in formal specification-based retrieval scheme, and thus improves retrieval reliability.
- Query construction for retrieval of desired classes is very easy. The concept lattice also allows retrieve engine to process queries more efficiently and provides optimal feedback to the user.
- Non-primitive data types and user-defined data types are allowed to choose from the list.

- Methodology of concept-based retrieval is language independent. Although we implemented the CBRS using Java, the methodology can handle programming languages with multiple inheritance equally well.
- The methodology allows abstract classes and interfaces to be included in the context that represents a class library.
- A multiset of set types representation of a class and incorporation of arguments of methods into context eliminate the need of knowing arguments order of the class' methods in advance.
- The context of a class library allows us to incorporate more attributes to a class, including both formal and informal attributes such as a formally specified index term that represents class behavior, a very good class description, that will also present to the user, which greatly enhances the usefulness of the hierarchy for browsing purpose. Incorporation of mixed informal and formal attributes of a class into the context also allows us to overcomes the incompatible problems of formal and informal methods (Boudriga, Mili and Mittermeir 1992).

5.4 Conclusion

The previous methodologies of type-based and execution-based rely on static properties associated with a component and dynamic execution of test program on a class component, respectively. In this thesis, we proposed a new methodology based on the formal concept analysis. Our methodology tries to take advantages of the type-based and execution-based schemes as well as the formal method while still keeps a balance of simplicity and complexity. To reduce the complexity we use variable access patterns of methods of classes in the class library as representations of class behavior. To keep the simplicity of query construction we adopt an incremental selection mechanism instead of arbitrary entries by the users. Retrieving classes is investigated by an incremental selection mechanism to rapidly narrow down the search space. Based on the methodology, a prototype system is designed and developed using latest Java technology

and distributed CORBA architecture. Many tools are provided by our CBRS to allow users to browse the class library, retrieve precisely the suitable components from the library, organize the context for the library, build the corresponding concept lattice, and even graphically control the server side applications. The core of concept-based retrieval system is the lattice structure that represents a class library. The lattice structure can be studied by algebraic means and offers remarkable insight into properties and structure of the original class library. It is the ordered structure that improves both retrieval efficiency and reliability.

The concept-based retrieval is divided into five major phases: constructing class library, organizing the class library into a single-valued context via conceptual scaling if necessary, building the concept lattice, formulating user queries, and satisfying the queries. The CBRS is designed to handle exact and approximate retrievals with sophisticated GUIs for the end users. An easy-to-use, simple yet still powerful browse tool is designed to allow users to travel through the space of the concept lattice for browse-based retrievals. The direct and as flexible as near free-style navigation of the lattice structure is based on the new data structure of Java/Swing --- **JTree**.

Based on the work carried out in this thesis, we can draw the following conclusions:

- The CBRS provides a new, very useful and powerful tool for software developments based on reusing object-oriented class components. The idea may also be effectively applied to design new generation of search engines for the world wide web and other similar environments.
- The concept-based retrieval scheme proposed in this thesis reduces retrieval complexity and cost yet still maintain high recall and precision in terms of class behavior. The key is to impose a structural order of all the classes in a class library. This can only be achieved through a many-valued context to a single-valued context and then a concept lattice representation for the class library.

- Mixing type and behavior of methods of a class actually makes this methodology a combination of type-based and behavior-based retrievals. Furthermore, this methodology is flexible enough to allow us to incorporate more class attributes in terms of formal and informal methods into the context, so the potential is very promising.
- The FCA-based methodology provides a way to organizes the user's specification into a query. The query formulation is incredibly easy. The query constructed by the user has clear semantics, thus the result is very comprehensible to the user. The methodology combines the exact semantics of formal methods and the interactive navigation possibilities of informal methods.
- This methodology also provides a way of an incremental selection from a list of available composite types by the user, which allows exact match and completely eliminates the data mismatch and time-consuming deduction phase in a formal specification-based retrieval scheme, and thus improves retrieval reliability. The concept lattice allows retrieve engine to process queries more efficiently and provides optimal feedback to the user.
- The CBRS system supports exact and approximate retrievals. The approximate retrieval makes it useful for retrieving components with similar class behavior to the user query.
- Browsing a class library is made possible based on two reasons: (1) the users may not be familiar with the retrieval system and have no concrete knowledge what they are looking for; (2) the users want to go through browsing a class library to achieve the purpose of retrieval of partially-satisfied classes because there is not always an exact match between the user's requirements and available classes in the class library. In both cases, the user wants to look around for a class that is close enough to be adapted to his/her needs. The hierarchy generated from the concept lattice of a class library can be used for browsing and understanding the relationships between the classes.
- Navigating directly on a concept lattice hierarchy of a class library actually plays the roles of both browsing and searching the library. The navigating is achieved through an applet called **ConceptLattice** that is implemented in Java/Swing **JTree** data structure. The Swing **JTree** is a perfect choice to display hierarchical data. The advantage of using concept lattice is that we can incorporate more attributes to a class like sample

behavior or a very good class description for each node (concept) that will also present to the user, which greatly enhances the usefulness of the hierarchy for browsing purpose.

- A distributed CORBA client/server applications implemented using Java and Java/Swing not only make its easy to put the whole system on the Internet and enterprise Intranet, which enable the user perform remote method calls, but also provides professional graphic user interfaces.

There are a few things left both in the methodology and implementation. They can be summarized as follows:

- As discussed early incremental selection of composite types from the list displayed by the CBRS narrows the search space rapidly. Similarly, the search space should be widened again by deselecting a composite type. Our methodology did not specify how to handle this situation although the user may remove an item from the query any time. To solve this problem, the system needs to calculate the new focus using a join operation instead of a meet operation.
- The retrieval style of our CBRS is attribute-based system: the focus is essentially a function of the selected attributes (composite types). Due to the its dual nature, concept lattice also allows object-based (class-based) retrieval style although it is more appropriate for library understanding and reengineering (Fischer 1998). Our methodology and implementation only allow attribute-based retrieval style. The solution is that the user selects or deselects a single class and system calculates the new focus accordingly. However, selecting an additional class widens the focus and is thus realized by a join operation.
- A tool of automatic line diagram generation is very useful for the system administrator. It is not implemented in this thesis.

5.5 Future Work

A number of ways exists in which the methodology and the implementation in our CBRS can be further extended.

The class maintenance of our methodology needs improvements. Currently, we have to modify the context, recalculate the concepts and rebuilt the concept lattice after an addition of a new class into the class library or a deletion of a class from the class library. This process is tedious and calculation-intensive. So adopting an incremental update algorithm such as the algorithm mentioned by Godin et al (Godin, Missaoui and Alaoui 1995) instead of Ganter's batch algorithm (Ganter 1986) would solve this problem. This requires an implementation of the incremental batch algorithm and perhaps best implemented with graphic user interface for the administrator to manage the CBRS.

Automatic drawing of line diagram is useful to the system administrator even for some batch algorithms. Bordat's algorithm (Bordat 1986) deals with building Hasse diagram for a context. An implementation of this algorithm will update our CBRS to include the automatic generation of concept lattices.

A simple extension of class-based retrieval style to the CBRS will make the system more general purpose. It is not only suitable for reuse purpose through the composite-type-based style, but also can be used for library understanding and reengineering through the class-based style because object-based navigation exposes implicitly conceptual similarities of classes (Fischer 1998).

References

[1] An, N. and Park, Y. 1998, A Structured Approach to Retrieving Functions by Types, Proceedings of the ACM International Conference on Functional Programming (ICFP).

[2] Atkinson S. and Duke, R. 1995, Behavioral Retrieval from Class Libraries, Austrian Computer science Communication, 17(1), 13-20.

[3] Bergstein, P. and Lieberherr, K. 1991, Incremental Class Dictionary Learning and Optimization, in Proceedings of the European Conference on Object-Oriented Programming, Geneva, Switzerland, Springer Verlag.

[4] Berzins and Luqi ,1991, Software Engineering with Abstractions, Addison-Wesley.

[5] Biggerstaff, T. J. and Perlis, A. J. 1989, Software Reusability, Vol. I and II, ACM Press.

[6] Birkhoff, G, 1993. Lattice Theory, 3rd edition, American Mathematical Society, Providence, RI.

[7] Booch, G. 1994, Object-Oriented Analysis and Design, with Applications, Addison-Wesley, Menlo Park, CA, Second Edition.

[8] Bordat, J. P. 1986, Calcul Pratique du Treillis de Galois diune Correspondance, Mathematique et Sciences Humaines, 96, 31- 47.

[9] Borster, J. 1995, The 7th International Conference on Software Engineering and Knowledge Engineering, Rockville, MD, Jun 22-24, 204-211.

[10] Boudriga, N., Mili, A. and Mittermeir, R. 1992, Semantic-Based Software Retrieval to Support Rapid Prototyping, Structured Programming, 13, 109 - 127.

[11] Chou, S. C., Chen, J. Y. and Chung, C. G. 1996, A Behavior-based Classification and Retrieval Technique for Object-Oriented Specification Reuse, Software --- Practice and Experience, 26(7), 815-832.

[12] Cole, R. J. and Eklund, P. W. 1996a, Application of Formal Concept Analysis to Information Retrieval using a Hierarchically Structured Thesaurus, International

Conference on Conceptual Graphs, ICCS '96, Sydney, 1-12, University of New South Wales, 1996.

[13] Cole, R. J. and Eklund, P. W. 1996b, Text Retrieval for Medical Discharge Summaries using SNOMED and Formal Concept Analysis, Australian Document Computing Symposium, 50-58.

[14] Cole, R. J. and Eklund, P. W. 1999, Analysing an Email Collection using Formal Concept Analysis, European Conference on Knowledge and Data Discovery, PKDD'99, at following URL: http://www.int.gu.edu.au/kvo/publications/cole.pdf.

[15] Coleman, D., Arnold, P., et al 1994, Object-Oriented Development: the Fusion Method, Englewood Cliffs, NJ: Prentice Hall.

[16] Cook, W. R. 1992, Interfaces and Specification for SmallTalk-80 Collection Classes, in Proceedings of the Conference on Object-Oriented Programming System, Languages and Applications, A. Paepacke (ed.), Vancouver, B. C., Canada, ACM Press, 1 -15.

[17] Cox, B. J. 1990, Planning the Software Revolution, IEEE Software, 7(6), 25 - 35.

[18] Damiani, E., Fugini, M. G. and Fusaschi, E. 1997, A Descriptor-Based Approach to OO Code Reuse, Computer, 30(10), 73-80.

[19] Davery, B. A. and Priestley, H. A. 1990, Introduction to Lattices and Order, Cambridge University Press, Cambridge, UK, 2nd edition.

[20] Davies, B. J., Automatic Class Hierarchy Design Using Formal Concept Analysis, The University of Newcastle Report, 1997.

[21] Dvorak, J. 1994, Conceptual Entropy and Its Effect on Class Hierarchies, IEEE Computer, 27(6), 59 - 63.

[22] Fischer, B. 1998, Specification-Based Browsing of Software Component Libraries, in Proceedings of Automated Software Engineering (ASE'98), Hawaii, 246 - 254.

[23] Fischer, B., Schumann, J. M. P. and Snelting, G. 1998, Deduction-Based Software Component Retrieval, in W. Bibel and P. H. Schmitt, editors, Automated Deduction - A Basis for Application, Kluwer, Dordrecht, 265-292.

[24] Funk, Lewien, and Snelting 1995, Algorithms for Concept Lattice Decomposition and Their Applications, Computer Science Report 95-09, Technical University of Braunchweig, Germany.

[25] Ganter, B. 1986, Algorithm for Formal Concept Analysis, in B. Ganter, R. Wille and K. E. Wolff, editors, Beitrage zur Beriffsanalyser, BI Wissenschaftsverlag, Mannheim, 241-154.

[26] Ganter, B. and Wille, R. 1999, Formal Concept Analysis – Mathematical Foundation (English Edition), Springer-Verlag, Berlin. Or visit following Web site for an introduction: http://www.math.tu-dresden.de/~ganter/concepts.ps.

[27] Ganter, B., Stahl, J. and Wille, R. 1986, Conceptual Measurement and Many-Valued Contexts, in Classification as a Tool of Research, W. Gaul, M. Schader, editors, Elsevier Science Publishers, North-Holland, 169-176.

[28] Godin, R., Missaoui, R. and April, A.1993, Experimental Compression of Navigation in a Galois Lattice with Conventional Information Retrieval methods, International Journal of Man-made Machine Studies, 38, 747 - 767.

[29] Godin, R., and Mili, H 1993, Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattice, in Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93), A. Paepcke (editor), Washington, D. C.: ACM Press, 394- 410.

[30] Godin, R., Mili, H., Mineau, G., Missaoui, R., Arfi, A. and Chau T. T. 1998, Design of Class Hierarchies Based on Concept (Galois) Lattices, Theory and Application of Object Systems, 4(2), 117 - 134.

[31] Godin, R. and Missaoui, R. 1994, An Incremental Concept Formation Approach for Learning from Databases, Theoretical Computer Science, 133, 387 - 419.

[32] Godin, R., Missaoui, R. and Alaoui, H 1995, Incremental Concept Formation Algorithms Based on Galois (Concept) Lattices, Computational Intelligence, 11(2), 246 - 267.

[33] Godin, R., Mineau, G. and Missaoui, R. 1995, Applying Concept Formation Methods to Software Reuse, International Journal of Software Engineering and Knowledge Engineering, 5(1), 119 - 142.

[34] Godin, R., Sanders, E. and Grecsei, J. 1986, Lattice Model of Browsable Data Spaces, Information Sciences, 40, 89 - 116.

[35] Henninger, S. 1994, Using Iterative Refinement to Find the Reusable Software, IEEE Software, 48 -59.

[36] Johnson R. and Foote, B. 1988, Designing Reusable Classes, Journal of Object-Oriented Programming, June/July, 22 - 35.

[37] Katz, S., Richter, C. A. et al 1987, A System for Reusing Partially Interpreted Schemas, Proceedings of 9th International Conference on Software Engineering, 377-385.

[38] Korson, T. and Mcgregor, J. D.1992, Technical Criteria for the Specification and Evaluation of Object-Oriented libraries, Software Engineering Journal, March, 85 - 94.

[39] Krone, M. and Snelting, G. 1994, On the Inference of Configuration Structures from Source Code, in ICSE-16 Proceedings of the 16th International Conference on Software Engineering, IEEE Computer Society press, Los Alamitos, California, 49 - 57.

[40] Liao, H. C. and Wang, F. J. 1993, Software Reuse Based on a large object-Oriented library, Software Engineering Notes, 18(1), 74-80.

[41] Lieberherr, K. J., Bergstein, P. and Silva-Lepe, I. 1991, From Objects to Classes: Algorithms for Optimal Object-Oriented Design, Journal of Software Engineering, 6(4), 205- 228.

[42] Lindig, C. 1995, Concept-Based Component Retrieval, Proceedings of IJCAI Workshop on Formal Approaches to the Reuse of Plans, Proofs, and Programs, Montreal.

[43] Lindig, C. 1998, Analysis of Software Variants, Slides for the Talk Given at the *Program Understanding and Software Reengineering* Seminar at Dagstuhl, Germany, March 9 -13.

[44] Lindig, C. and Snelting, G. 1997, Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis, Proceedings. of International Conference on Software Engineering, 349-359.

[45] Maarek, Y., Berry, D. and Kaiser, D. 1991, An Information Retrieval Approach for Automatically Constructing Software Libraries, IEEE Transaction on Software Engineering, 8(17), 800-813.

[46] McManis, C. 1996, Java in Depth, http://www.javaworld.com/jw-12-1996/jw-12-indepth.html.

[47] Mili, A, Mili, R. and Mittermeir, R. 1997, Storing and Retrieving Software Components: A Refinement-Based System, IEEE Transaction on Software Engineering, SE-23(7), 445-460.

[48] Mili, A, Mili, R. and Mittermeir, R. 1998, A Survey of Software Reuse Libraries, Annals of Software Engineering.

[49] Monarchi, D. E. and Puhr, G. I. 1992, A Research Typology for Object-Oriented Analysis and Design, Communications of ACM, 35(9), 35 - 47.

[50] Nelson, M. L. and Poulis, P. 1995, The Class Storage and Retrieval System: Enhancing Reusability in Object-Oriented Systems, OOPS Messenger, 6(2), 28-36.

[51] Niu, H. J. 1999, Execution-Based Retrieval of Object-Oriented Components for Reuse, MS Thesis, School of Computer Science, University of Windsor.

[52] Niu, H. and Park, Y. 1999, An Execution-based Retrieval of Object-Oriented Components, Proceedings of the ACM Southeast Conference (ACMSC), 1999.

[53] Oosthuizen, G. D., Bekker, C. and Avenant, C. 1992, Managing Classes in Very Large Class Repositories, in Proceedings of the Tools, Paris, 625 - 663.

[54] Opdyke, W. and Johnson, R. 1989, Refactoring: An Aid in Designing an Application Framework, in Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications, 114 - 118.

[55] Opdyke, W. and Johnson, R. 1993, Creating Abstract Superclasses by Refactoring, in ACM Computer Science Conference, 335- 339.

[56] Park Y. 1999, Software Component Retrieval by Samples using Concept Analysis, Proceedings of the IASTED Conference on Applied Informatics, 1999.

[57] Parnas, D. 1994, Software Aging, in ICSE-16 Proceedings of the 16th International Conference on Software Engineering, IEEE Computer Society press, Los Alamitos, California, 279 - 290.

[58] Pennix, J., Baraona, P. and Alexander, P. 1995, Classification and Retrieval of Reusable Components Using Semantics Features, in Proceedings of 10th Knowledge-based Software Engineering Conference, Boston, MA, Nov. 1995, IEEE Comp. Soc. Press, 131-138.

[59] Perry, D. E. 1987, The Inscape Environment, in Proceedings of 11th International Conference on Software Engineering, IEEE Comp. Soc. Press, 2-12.

[60] Prediger, S., 1997, Logical Scaling in Formal Concept Analysis, in Conceptual Structures: Fulfilling Peirce's Dream, D. Lukose et al, editors, Proceedings of the ICCS'97, LNAI 1257, Springer, Berlin, 332-341.

[61] Prieto-Diaz, R. and Freeman, P. 1987, Classifying Software For Reusability, Software, 6-16.

[62] Prieto-Diaz, R. 1991, Implementing Faceted Classification for Software Reuse, Comm. ACM, 34(5), 88-97.

[63] Richards, D. and Compton, P. 1997, Knowledge Acquisition First, Modelling Later, Proceedings of EKAWA'97, Sant Feliu de Guixol, Spain, October 1997, Springer, Berlin.

[64] Rollins, E. J. and Wing, J. M. 1991, Specification as Search Keys for Software Libraries, in K. Furukawa, editor, Proceedings of 8th International Conference on Symp. Logic programming, Paris, June 24-28, MIT Press, 173-187.

[65] Rubin, K. S, and Goldberg, A. 1992, Object Behavior Analysis, Communications of the ACM Transactions on Software Engineering and Methodology, 3(2), 166 - 199.

[66] Salton and M. McGill 1983, Introduction to Modern Information Retrieval, McGraw-Hill, New York, N. Y.

[67] Schumann, J. M. and Fischer, B. 1997, NORA/HAMMER: Making Deduction-Based Software Component Retrieval Practical, in M. Lowry and Y. Ledru, editors, Proceedings of 12th International Conference on Automated Software Engineering, Lake Tahoe, Nov. 1997, 246-254.

[68] Shen, Y. and Park Y. 1999, Concept-based Retrieval of Classes using Access Behavior of Methods, 1st International Conference on Information Reuse and Integration (IRI-99), Atlanta, Georgia, 1999, 109 - 114.

[69] Siff, M. and Reps, T. 1997, Identifying Modules via Concept Analysis, Proceedings in International Conference on Software Maintenance, 170-179.

[70] Snelting, G. 1996, Reengineering of Configurations Based on Mathematical Concepts Analysis, ACM Transaction on Software Engineering and Methodology, 5(2), 146-189.

[71] Snelting, G. 1998, Concept Analysis - A New Framework for Program Understanding, ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, 1-10.

[72] Snelting, G. and Tip, F. 1998, Reengineering Class Hierarchies Using Concept Analysis, Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering, 99 - 110.

[73] Wille, R. 1982, Restructuring Lattice Theory: an Approach Based on Hierarchies of Concepts, in Ordered Sets, I. Rival (Ed.), 445-470.

[74] Wille, R. 1984, Line Diagrams of Hierarchical Concepts System, Int. Classif., 11(2), 77 -86.

[75] Wille, R. 1992, Concept Lattices and Conceptual Knowledge Systems, Computer and Mathematics with Applications, 23 (6-9), 493 - 515.

[76] Yau, S. S. and Tsai, J. J. 1987, Knowledge Representation of Software Component Interconnection Information for Large-Scale Software Modification, IEEE Trans. Software Eng., 13(3), 355-361.

[77] Zaremski, A. M. and Wing, J. M. 1995, Specification Matching of Software Components, in G. E. Kaiser, editor, Proceedings of 3rd ACM SIGSOFT Symp. Foundation of Software Engineering, Washington, D. C., October 1995, 6-17.

Vita Auctoris

Name: Yu Shen

Place of Birth: Shandong Province, China

Year of Birth: 1963

Education:

1980 - 1984 B. Sc., Department of Physics, Qufu Normal University, Qufu, Shandong, China

1984 - 1990 Ph. D., Institute of Applied Chemistry, Chinese Academy of Sciences, Changchun, Jilin, China

1994 - 1998 Research Fellow and Network Support Specialist, Department of Materials Science and Engineering, University of Michigan, Ann Arbor, Michigan, USA

1998 - 2000 M. Sc., School of Computer Science, University of Windsor, Windsor, Ontario, Canada