The University of New South Wales
School of Computer Science and Engineering

Class-Library Management System for Object-Oriented Programming

Gi-Moon Nam

supervised by
Dr. Jian Ma

18 credit project report
submitted as a partial requirement of M.Comp.Sc.
March, 1993

Acknowledgements

I would like to express my sincere thanks to my supervisor, Dr Jian Ma, for his patience, serious doubts and constructive criticism in the development of this thesis.

To my fellow students at UNSW, particularly Kek-Wee and Banchong, for the stimulating discussions on all kinds of topics. Thanks also to the staff of 'English Supporting Unit' and Grace who helped with my English.

Finally, my deepest thanks go to my parents, who have given me the love and financial as well as emotional support without which this thesis could not have been written.

Abstract

Object-oriented (OO) software development has been touted to promote and enhance the reuse of existing software. This advantage, together with its emphasis on concepts such as data abstraction, information hiding, encapsulation and inheritance, provides significant advantages as it allows software developers to tackle complex problems several orders of magnitudes higher than existing programming technologies can afford. However, if this ability to manage complexity is to become a reality, an OO development environment must learn to manage its class library efficiently so that its developers can reuse the class information in the library. The problem becomes particularly serious when the number of classes increases to a certain level in an environment such that retrieval of information on existing classes becomes impossible. Current techniques for managing large class libraries make use of browsers. However, these tend to impose severe limitations when the size of the library increases as our experience with developing a library of collection classes using C++. This prompted us to look into an alternative approach for tackling the management of large class libraries via a relational data dictionary approach, where data dictionaries are used to describe the information about concepts in C++ and are then implemented using a relational database management system, INGRES. The proposed technique is capable of providing the developer with an efficient and non-procedural method for retrieving information in a class library.

Contents

Chapter 1

Introduction

Object-oriented programming has become popular in recent years for the development of a variety of application software systems. It emphasises the concepts of data abstraction, information hiding, encapsulation and inheritance, and provides advantages such as natural decomposition of the application problem domain, software reuse, and modularity.

Object classes are the basic concepts of software development using object-oriented programming languages such as C++ [Str91], Smalltalk [GR83], and Eiffel [Mey92]. By serving as a key component which implements the major concepts of object-oriented programming, classes are a kind of valuable resource that needs to be effectively managed.

Current approaches for object-oriented software developments make use of the browser technique for the management of classes in object-oriented programming. However, these methods encounter several major problems that limit the use for a large library system in a multi-user distributed software development

environment.

This research attempts to propose a 'class library management system(CLIMS)' which utilises existing relational database management systems(RDBMS). By use of the CLIMS, the object classes developed by individual programmers are centrally managed and users are provided with facilities for locating and retrieving classes they need and comprehending functionalities of the retrieved classes.

1.1 Background

1.1.1 Object-Oriented Programming

Object-oriented programming is a new methodology for software development which encourages the use of modern software engineering technology and promotes software reuse. However, in technical literature such as [CW85] and [Mey88], there has been no common definition of what object-orientation is, although this problem has been addressed [WBJ90].

The key concepts that describe object-oriented software development are data abstraction, information hiding, encapsulation and inheritance. They are defined as follows [NMN93]:

- **Data abstraction** has its roots in modular programming. It allows the behaviour of a real world entity to be defined by a set of abstract operations. These abstract operations emphasise the essential characteristics while suppressing implementation details that are temporarily unimportant. The result is to make the system less susceptible to changes in implementation as

long as the external interface remains unchanged.

- **Information hiding** is a technique through which certain inessential details of an item are made inaccessible. By providing only essential information, it achieves the goal of reducing coupling by keeping things simple and limiting unintended interactions. This reduces the chances of system corruption since the only interaction between the class and its client is through their interface.

- **Encapsulation** is a term frequently confused with information hiding. It actually refers to the act of enclosing the abstractions as syntactically and semantically bound units.

- **Inheritance** is a technique for the reuse of classes. Since the interface of a class serves as a contract between specification and its implementation, inheritance then allows a new class to be defined by making use of the existing ones. Inheritance in object-oriented development takes two forms, namely: specialization, which uses generalized functions in the superclasses; and incremental modification, which takes the original functions of the superclasses and extends the functions to suit more general usage.

Object-oriented systems are built as collections of classes [Mey88] each of which serves as a key component in object-oriented languages. A *class* is a language construct that is an abstract data type implementation. Based on the class, encapsulation and information hiding is also implemented, since the two concepts are expected to be built around the data abstraction. Additionally, as a key for reusability and extendability [HHKM91], the class serves as a template from which objects can be created [Weg90]. Two basic approaches to reuse classes are construction and subclassing: encapsulation and inheritance support the two mechanisms respectively [DMS89]. The class itself can be reused by defining new

classes which create the instances of existing classes. Or, with inheritance, both single and multiple, a new class(derived class) inherits what it needs from a existing class(base class). Encapsulation makes some valuable code or implementation detail invisible and provides the interface which is only of interest to reusers.

A large number of programming languages that support object-oriented programming features have been developed and are being used for software development: examples include Smalltalk, C++, Eiffel, etc.. In this research, C++ has been chosen as our language of use because:

1. C++ is an extension of C, which is one of the most widely used programming languages for industrial and commercial applications. That means a large number of potential users of C++. Thus the need for a class library management system is strong.

2. C++ supports a rich set of object-oriented features and also provides a very clean object-oriented model to work with.

3. Environments using the C++ programming language include not only basic classes from the compilers, but also add-on classes developed in-house or purchased from commercial vendors. The management problem is much more evident since there is a larger pool of class libraries to manage.

4. C++ has been widely used in developing software applications in school laboratories, but it lacks tools for supporting users to access and reuse the classes in the library efficiently.

1.1.2 Software Reuse

Agresti [AM88] defines the software reuse as employing knowledge that has been compiled through previous experience: the term 'past experience' includes requirements, specifications, modules, designs and software components. There are three types of software reuse in software development [BBJR87]: informal knowledge, schematized knowledge and productized knowledge.

It has been believed that the reuse of software could improve its productivity by increasing the programmer's productivity, and provide reliable software by increasing software quality, because reusable software components must have gone through a strict process of validation and verification. The improvement in reliability also means cheaper expense in maintenance, which consumes the largest portion of system costs. Other contributions of software reuse are consistency through the same component in many places, manageability through the use of well-understood components, and standardisation through the use of standard components [AM88].

However, this espoused principle is still low in practice because of both technical and non-technical problems [Mey87] [AM88] [WI88]. To succeed in reuse, a well organised reuse program needs to integrate many factors from different technologies, process models and cultures.

Frakes [FPDMS91] suggested major technologies that enable reuse:

- **Libraries:** The value of library technology lies largely in establishing a concrete process infrastructure that fosters reuse by its existence.

- **Object-oriented programming languages:** The main value is in the perspicacity of the representation and its tendency to foster larger and more abstract reusable components (i.e., classes and frameworks) than in earlier languages (i.e., functions). Furthermore, an object-oriented representation tends to lead to clearer, more elegant and compact designs.

- **Classification schemes:** The main value of classification schemes is that these schemes force the issue of understanding the problem and application domain.

In this thesis, the two technologies, libraries and object-oriented programming languages, will be primarily focused on, in order to show how the class-library management system for object-oriented software development can realise the reusability of software components. The classification scheme will be also considered in organising the class library.

1.1.3 Class Library Management

Class-libraries are repositories of classes that serve as reusable building blocks for object-oriented software development [Weg90]. Again, the class declaration is a key unit which supports reusability and expandability. All of the object-oriented features are implemented based on the class declarations. Therefore 'class library management' is necessary to exploit the features of object-oriented software development, especially the reusability and expandability of the existing software components. It is widely held that in order to effectively promote the reusability of software components, the tools that manage the library of reusable software components (classes in object-oriented languages) are necessary to help the reuser to find candidate components for reuse efficiently[WB87][GTNP90]

[AS87][Mey87][KM92][SEHVG91]. Additionally, class management should include those functions such as data modelling, access methods and authorisation, class packaging, and security control.

Building class library management systems by utilising information system seems desirable since there are many advantages to managing classes within an information system[GTNP90]:

- It eases the retrieval process through indexing or keywording scheme.
- Software developed using the existing classes in the library has high reliability through the application of the quality control procedures on the classes added to the library.
- By managing the knowledge about relationships between classes, the system can help the reuser to understand the classes.
- By obtaining a class from a centrally managed library system, developers are more likely to get a standard version rather than a version full of undocumented local modification.

1.1.4 Ideal Scenario of Software Development

The scenario of software development under a well matured reuse environment will be:

> A library of reusable software components which are developed by a software community familiar with the application domain is managed. To develop a new application, a developer searches the library to find

candidate software components which satisfy the given set of requirements. If the components that satisfy all requirements are available, reuse becomes straightforward by just initializing and composing the selected components to construct the application. But usually, the candidate components from the library do not satisfy the requirements completely. In this case, the developer chooses the components that require the least adaptation to match the requirements with the help of the library management system, and adapts them to build the application.

1.2 Objectives of the Research

The objectives of this study comprise the following:

1. To review current approaches to the management of reusable libraries in different programming environments;
2. To design a class library management system(CLIMS) using a relational data dictionary approach, where data dictionaries are used to describe the information about classes in C++;
3. To implement CLIMS by making good use of an INGRES system;
4. To test the practicability and applicability of CLIMS through a case study.

1.3 Scope of the Research

This research aims at the development of a class library management system(CLIMS) to facilitate software reuse in object-oriented programming. The prototype system of CLIMS which has been proposed in this thesis includes the following features:

- A tool which extracts reuse-information from C++ classes and populates class libraries with the reuse-information.
- A tool which supports the location and retrieval of reusable classes.
- A tool which help reusers to understand the retrieved classes for reuse.

CLIMS is developed by using the INGRES [SKWH76] package (e.g. SQL, 4GL and RBF) and embedded-C for the management of C++ classes under the Apollo Domain/OS(SR10.4) environment. Thus the proposed system is workable where:

1. The software application is developed using C++ under UNIX environment.
2. There is a relational database management system INGRES 6.02/03 packages (SQL, 4GL, ABF and RBF) in a network environment.

However, the proposed system does not include:

- A complete implementation of a user interface,
- A tool to support the management of other software components apart from classes in C++, and
- A tool to support the evolution of class libraries.

1.4 Thesis Organization

In chapter 2, the current research of software library management systems are categorised according to their approaches and reviewed. The approaches include a browser, an information retrieval (IR)-related and an artificial intelligence (AI)-approach.

In chapter 3, the design of CLIMS is presented. First, the system requirements are defined. Second, several issues which are relevant to the design of CLIMS are discussed. Finally, the design of a data structure and functions of CLIMS are proposed.

In chapter 4, the implementation of CLIMS is presented. The data structure and functions which are introduced in chapter 3 are implemented one by one.

In chapter 5, the applicability of CLIMS is demonstrated through a case study.

In chapter 6, the conclusion and further research directions are presented.

Chapter 2

Literature Review

There are four major issues that need to be taken into consideration for designing a library management system which supports the reuse of software components:

1. How to classify software components in order to organise the software library.
2. How to represent the reusable attributes of software components inside storage.
3. How to support searching and retrieving of reusable components.
4. How to design a user-friendly interface to allow users to access the libraries conveniently.

Currently, several library management systems have been proposed and developed to facilitate software reuse in different programming environments. They are surveyed to gain an insight of what are the minimal requirements or features of a library management system and how the issues mentioned above can be realised in such systems.

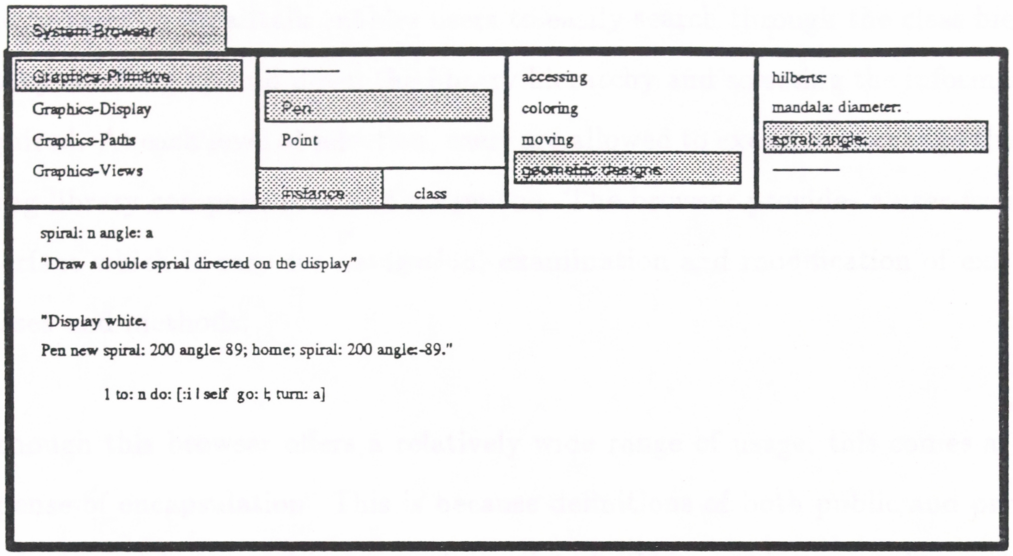


Figure 1: System Browser window

2.1 Library Management System using a Browser Approach

2.1.1 Browser in Smalltalk-80

The browser in Smalltalk-80 [GR83] is a class library management system attached to the Smalltalk programming environment. It arranges the classes in a hierarchical structure and provides the end-users with a graphical and textual interface. Figure 1 shows an example of the browser in a Smalltalk system, where there are views for the class 'pen' and its message selector[1] 'spiral:angle' are shown, the categories of which are displayed on the first and third subviews, respectively.

[1]the message selector is a Smalltalk terminology which corresponds to the member in C++

The browser in Smalltalk enables users to easily search through the class hierarchy by scrolling up and down the library hierarchy and selecting the information required. At each level of selection, users are allowed to examine or change the existing library component or add a new one. The browser provides a user-friendly interface which is easy for navigation, examination and modification of existing classes and methods.

Although this browser offers a relatively wide range of usage, this comes at the expense of encapsulation. This is because definitions of both public and private methods are shown and can even be updated. This problem may decrease the reliability of the library because the quality of the library hinges on the quality of its users. The views supported by the library could be different depending on the user's perspective. The clients of the library are supposed to know only the interface of a class and not the implementors' view as well. In short, the browser does not support the abstraction of data types and encapsulation which contradicts the principles of object-oriented programming [Wu90]. Another shortcoming of the browser arises when the number of classes in the library increases, in which case browsing through the inheritance tree may not reveal the candidate component quickly. In addition, there could be a case when classes unrelated by inheritance but yet providing the required functionality fail to be revealed because of the inherent search technique used [GTNP90].

2.1.2 Eiffel Library System

The Eiffel library is a collection of classes in the Eiffel language that is an object-oriented programming language developed with strong support of reusability in mind. Standard Eiffel delivery includes class libraries (categorised by application

domains) and a graphical browser as a library management tool. The browser makes it easy to obtain any information about all the classes in a set of categorised directories. The operations through the browser is basically similar to that in Smalltalk.

The Eiffel library system supports strong reusability of software components mostly attributed to the strength of the inherent language design [Mey90]. All routines and classes in Eiffel are associated with formally specified *assertions* which are a language construct for helping the construction and use of libraries. To support the understandability of classes during a reuse process, the Eiffel library extracts the documentation as much as possible from the class texts. This extraction of documentation from the source code improves the consistency and reliability of reuse information. A *Flat-short form of a class*[Mey88] which shows the interface properties, is automatically produced by the support of the structure of the language. This interface not only provides functional interface (black-box view) but also relieves a user of the traversal of the ancestor path to find out if there is a function in one of the ancestors to overwrite [Wu90]. Eiffel also provides a language mechanism, *obsolete*, which helps the library designer or maintainer remove obsolete features progressively without affecting the client classes. In addition to the browser, a facility which allows classes to include *indexing* for retrieval is provided by the language mechanism,

However, the current library system has not yet addressed the situation when the size of the class libraries increases. Meyer [Mey90] also mentioned the requirement of advanced query supporting. Basically, the Eiffel library system shares the problems of the Smalltalk library, which result from the browser itself. In addition, because many techniques which support the reusability of classes in the Eiffel library management system come from the language specific features,

it is difficult to apply them to other libraries.

2.2 Library Management System using an Information Retrieval (IR) -Related Approach

2.2.1 RSL System

The Reusable Software Library (RSL) [BRB+87] has been developed to promote reusability by Intermetrics. Its prototype caters mostly to Ada, which supports reuse through its package and generic features, but it also caters to software written in other languages. Figure 2 shows an overview of the RSL system. The architecture of the RSL consists of the RSL database and four subsystems: Library Management, User Query, Software Component Retrieval and Evaluation(Score), and Software Computer-Aided Design(SoftCAD) subsystem.

The 'Library Management Subsystem' is used to extract explicitly labelled reuse information automatically from the design or source code files, ensuring the quality of the extracted information, storing the qualified information into the RSL database and maintaining the RSL. Users can search for components using a *menu-driven* interface or *natural language* with the help of the User Query Subsystem. Score helps users to choose the most appropriate component to reuse among many candidate components by evaluating the stored components against the user's software requirements. A user can do a high-level design of software with SoftCAD which provides a graphic design and documentation tool. Because SoftCAD has been integrated with the RSL database, users can retrieve the components from the RSL to use in a SoftCAD design, and can enter new components

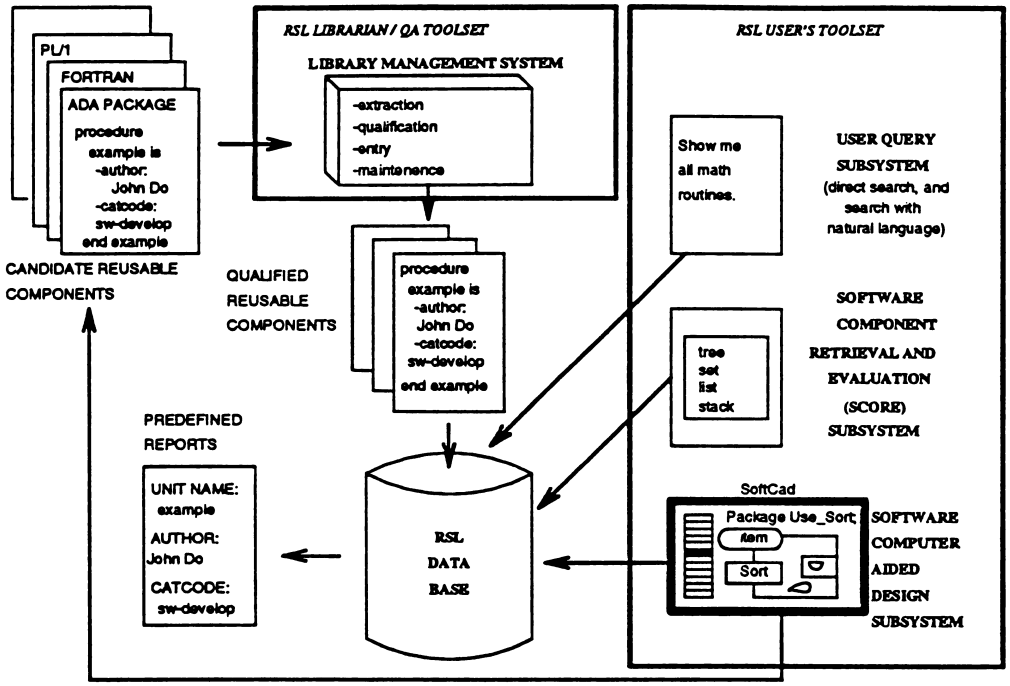


Figure 2: Overview of the RSL prototype

written as a result of a SoftCAD design into RSL.

Searching for the reusable components with easy-to-use queries(User Query Subsystem) and selecting the most appropriate component to reuse among the candidates with the interactive evaluating system(Score) are the core parts of the RSL.

However, the system suffers from the tedious need to manually insert the reusable statements to the PDL or source code file, as well as to ensure the quality of software components for entry into database.

Since the system allows several types of language components, it cannot exploit the mechanisms of those languages that support reuse. Object-oriented language features that facilitate reuse are not integrated into the system. As a result, the system does not provide a reliable and functional view of a component that is

essential to show the usage of the component. Users still need to depend on the documents written by the librarian or read through the whole code for themselves.

2.2.2 The REUSE System

The REUsing Software Efficiently(REUSE) system [AS87] is proposed to encourage the reuse of software components by providing a cataloguing and retrieval system. The system provides a menu-driven front-end to an IR system, through which existing reusable software can be catalogued and stored systematically, and can be easily located for reuse. The information retrieval(IR) system is employed over the database management system(DBMS), because it is thought that the type of information that is mainly dealt with in the REUSE system is textual.

The user may specify all known information which is necessary to find the software components to be reused. Here the system provides flexibility for users to express their information through both user-menus and free-form words.

Flexible user queries (free-form and menu-driven) for the retrieval process and efficient access to the required information are the advantages of this system, whose capabilities are due to the proper use of the IR system's nature. On the other hand, the use of IR system creates limitations of the system. It is doubtful that the REUSE system can work satisfactorily, especially when dealing with the evolution of the organisation, if the amount of data to be managed increases. This is one of the natural disadvantages of the inverted-index scheme which the IR system is based on. The integrity and consistency of the system is another thing to be questioned, since descriptive information for reusable components is not from the software itself but from humans. Additionally, lack of facility

that provides functional interface will burden reusers with having to grasp the functionality of a component.

2.2.3 GURU system

The GURU system [MBK91] is for automatically constructing large software libraries which promote software reuse. The library construction procedure consists of two stages, indexing and classifying software components. First, indices(attributes) which include conceptual information about the functionality of each component, are automatically extracted from the natural-language documentation. Second, the components are classified into clusters such that the members of the same cluster share the same set of properties.

A new indexing scheme based on the *lexical affinities* has been introduced for the GURU. The basic principle of 'lexical affinities' is that if two words, say A and B, appear together in a same sentence frequently, the words A and B have a high correlation between them. Using this indexing scheme, the system reads through the documentation of each component and extracts indices automatically. Because the system cannot distinguish the meaningful from the useless (low-level, for example comments for variables) documentation, it has chosen manual pages for each software component as the target documentation to be processed. The indices built in the indexing stage are stored into an inverted-index file to allow fast retrieval of candidate components which fully or partially match the user query.

Finally, a clustering technique called *hierarchical agglomerative clustering* was applied to software components and a browse hierarchy was built where each

node is a set of software components with functional commonality.

A typical procedure for user interaction to retrieve a component is the following:

- User inputs his query in the natural language.
- Applying the same technique used in the indexing stage, the system receives the attributes from the user query, whose attributes are comparable to the indices in the system.
- The system starts matching attributes from the user query against the indices, and displays the list of candidate components ordered by the degree of matchability.
- If the user does not satisfy the components found in the above process, the user browses through the hierarchy which was built at the classifying stage.

As the developers of GURU have also stated, automatic indexing by extraction of conceptual attributes from natural language documentation obviously gives more cost-efficiency and consistency over indices, which also makes it possible for the users to express their needs in the same natural language as the documentation is written in. Automatic classification using the clustering technique also is a strong point in that it helps users to browse through the relevant components, and so improves the retrieval efficiency in respect of *recall*[2]. In a word, the fully automatic building of library and natural language user queries are the core part of this system.

[2]recall is defined as the proportion of relevant materials; i.e., it measures how well the system retrieves all the relevant components

The choice of documentation which made automatic indexing possible is arguable since every reusable component cannot be expected to have manual-page-like documentation. Additionally, the classification scheme called clustering, which is based on the idea that the number of common words for describing two components determines the functional closeness for two components, is also arguable. Some components in the same domain possibly have fewer common words in the documentation that describe the components than those in different domains.

2.2.4 CATALOG Based System

The system discussed here has been developed using the existing information retrieval system, CATALOG [WB87] to store and retrieve software components. Each component is identified by a set of single-term indices that are automatically extracted from the natural-language documentation.

The system attempted to make the best use of the features of the CATALOG system. The CATALOG system allows a user to create, maintain, and search a database with both a menu-driven, command-driven mode or a boolean combination of search terms. Another feature of the CATALOG system is the supporting of the partial matching technique that increases the recall of the retrieval process. Inverted indexes contain every meaningful word in a database and are used for searching. Two example user queries are the following :

```
Look for: sorting routines
Look for:((sorting and routines) or quicksort) not heapsort
```

The system prompts for queries with the phrase "Look for:". A user can express his requirements using the natural language (as in the first line in the example) or

using a boolean search expression through menu selection. Specifying a boolean expression in a query is also supported. In response to the queries from the user, CATALOG will find every possible related word in the database and display them with the number of related records for each element of the list, which will help the user to choose a suitable component.

The value of this system is that it employs the existing system, CATALOG, to build a software library and make good use of the features of the CATALOG system. Flexible user queries and high recall of the retrieval are other advantages.

However, the limitation of the IR system applies to this system also. The management of a large database and the way of representing the meaning of a text or other records in a way comprehensible to a computer are problems to be solved [WB87]. Another problem is the lack of a classification scheme to allow browsing through the functionally related components.

2.3 Library Management System using an Artificial Intelligence Approach

2.3.1 Component Description Frames Based System

The software library management system [WI88] discussed in this subsection borrowed the idea of natural language processing for the management of software components. The major types of software components which this system manipulates as a reusable unit are function, procedures and value-returning procedures,

and abstract data types[3].

The overall system can be categorised by their functionality: cataloguing system, component description frames and user interface. A catalogue of reusable components is built and it is represented by *component descriptor frames* inside the information base. When a user provides his request, the system shows the proper frames with blank slots according to the semantics of the request. After a user fills the slot(s), the system starts the matching process between the frame built from the request and the existing component descriptor frames, and shows the lists of candidate components ordered by the matchability.

The concept of *conceptual dependency* [WI88] is simplified and used for the *component description frames*. The basic idea is that the semantics of any software component description can be captured through a number of specifically related(dependency) fundamental concepts : objects, actions and modifiers. The component description frames are constructed for each basic function such as control, print, communication etc.. The frames for certain basic functions and the corresponding UNIX commands in figure 3 illustrate the above discussion.

Using the above technique, primitive software components which can be characterized by a verb, or an object on which a verb acts, may be dealt with efficiently. To handle an abstract data type such as a class in C++, a method which considers the abstract data types to be collections of primitive components is employed. So the abstract data type is stored in the system as a collection of component description frames, each of which represents each type of member function. Finding one of these functions allows the entire collection to be retrieved.

[3]The abstract data type is an abstraction over a type such as a 'class' in C++.

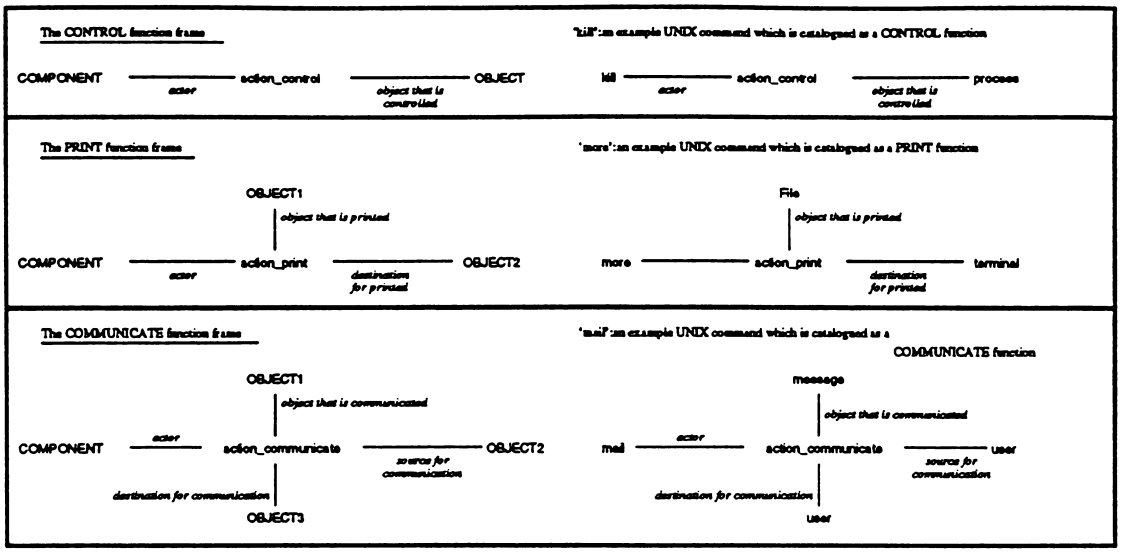


Figure 3: Examples of component description frames and corresponding UNIX commands

This retrieval technique may contribute to the improvement of recall and precision[4] by using the mechanism which tries to understand the semantics of the component description and user requirement.

While the system understands the semantics of the component description, it unfortunately ignores the semantics of the component itself: some components which are classified as being of the same category due to the same semantics of words that characterise those components, could be different in respect of their application domain or functionality. This drawback makes this retrieval technique only applicable to the domain that lacks a well-defined, generally accepted terminology [WI88]. Cost for developing descriptor frames is another problem. Lastly, the fact that the representation of software components (component description frame) in a library is always built around a verb reveals a limitation

[4]precision is defined as the proportion of retrieved material which is relevant; i.e., it measures how well the system retrieves only the relevant components

to cope with the software component that is characterised by a noun because of their inherent mismatch.

2.4 Summary of Literature Review

In this chapter, current software library management systems are categorised according to their approaches: a browser approach; an IR-related approach; and an AI approach.

The brief summary of the reviewed library systems with respect to the design issues is shown in figure 2.4.

The library systems using a browser approach discussed in this chapter are for maintaining classes in object-oriented programming languages, Smalltalk and Eiffel. They support an integrated software developing environment using existing classes, especially generic classes which are designed for future reuse. The graphic interface employed by the browser approach is easy-to-use. This interface could help with locating reusable classes easily by browsing through the class hierarchy. The problems of library systems using the browser approach are summarised as follows:

- The navigational approach of the browser makes it difficult for a software developer to retrieve a class in the class library quickly and efficiently if it has a large class repository.

	classification	navigational technique	user interface
Smalltalk Browser	application domain, class-hierarchy	hypertext	graphic browser window
Eiffel Browser	application domain, class-hierarchy	hypertext	graphic browser window
RSL	hierarchical category code attached to each component and descriptive keywords	query with index	menu-driven or natural-language query
REUSE	hierarchically organised menus and keywords	query with index	menu-driven or natural-language query
GURU	semantic similarity between indices	query with index	natural-language query or browser window
CATALOG using system	N/A	query with index	natural-language or menu driven query
AI approach System	semantics of words used to describe the software components	query with component description frame	forms-based

Figure 4: Brief summary of reviewed systems

- The users cannot query the whole library to look for specific features such as class types, inheritance relationships and so on. Instead, the user must navigate through the library, using the relationship links.
- Browsers use procedure-oriented techniques to retrieve information in a class library; thus complex navigational paths need to be defined by the user before retrieving a class.
- The technique does not support the concept of encapsulation.
- Navigation of a browser based on inheritance hierarchy does not provide any insight to the functionality of a class.

A software library system using the AI approach is generally more intelligent than other approaches such as the IR-approach, because it tries to understand the functionality of the software components and the semantics of queries. This intelligence may result in improvement in the efficiency of retrieval with respect to recall and precision. The system which we reviewed has borrowed the concept of conceptual dependency from the idea of natural-language processing in order to understand the semantics of software component descriptions. The problems of the library system using the AI approach are summarised as follows:

- It requires a high cost for domain-analysis or pre-encoding semantic information, which is essential since the intelligence in the AI approach mostly comes from the knowledge-base which contains semantics of domain information or natural language [MBK91].

- It lacks a classification scheme that reflects user requirements: application domain or functionality of software components[5].

The library management systems using an IR-related approach are generally cost-effective since automatic library building is possible using an indexing scheme in the IR system. The index scheme also allows fast retrieval of software components. Moreover, they usually support a flexible user interface such as natural language, keyword querying, and menu-front end interface.

The general problems with the IR approach are as follows:

- They lack reliability: For automatic indexing, they extract reuse attributes for components from document-style descriptions (e.g. GURU uses a manual page and CATALOG uses a header) not from the source code.

- They lack a classification scheme: since they usually extract any meaningful words from the text and build indexes for the components, some conceptual information for grouping is ignored[6]. Application domain and relationship between classes are types of information that should be considered in organising libraries.

- They lack in their support of large libraries: When the number of software components to be managed increases, the IR approach based on an indexed file cannot cope.

[5] Wood and Sommerville's classification scheme [WI88] is based on the ideas derived from natural language processing where semantics of a component description is the key for classification

[6] However, RSL supports classification through categories and keywords

- They lack support of encapsulation and abstraction that is essential when actually reusing the components: The user should depend on the documentation or read through the whole code to get to know how to reuse the components.

Another possible technique for a library management system is a 'formal specification'. A library is prepared with formally specified software components, and when the user inputs a formal specification of his requirements, the library system retrieves the candidate software components whose specification satisfies the requirements. Podgurski [PP92] mentioned this difficulty in practice since the current state of theorem-proving technology does not allow it.

As an alternative for these existing library management systems, a class library management system which uses the relational dictionary technique is proposed in this thesis. The proposed system is expected to supplement the existing library management systems, and to make the existing class libraries which lack a management tool more usable.

The relational database management systems are employed for an information base of our system, named CLIMS(Class LIbrary Management System). Some articles [AS87][WB87] argued that DBMS are not suitable for managing the information about reusable software components because of their weakness in dealing with an unformatted text: here, their assumption is that the reuse information processed by the information system is unstructured.

However, in CLIMS, the reuse information to be dealt with by the information

system could be structured because of the following reasons:

- The range of software components that are managed by CLIMS is narrow, only the classes.
- The source of reuse information in classes is the code itself rather than the unstructured documentation pages.

Employing RDBMS results in advantages such as the following:

- By using RDBMS, a lot of well matured techniques can be utilised for our purpose, especially for managing the distributed software development environment : eg. security control, query optimisation, consistency checking, etc..
- The cost for developing library management system can be reduced since most systems already have the RDBMS installed in it.
- RDBMS can cope with the evolution of the class library more efficiently than IR-systems: because of the overhead for updating the indices, the IR systems are limited in their ability to manage the collections of information which are very large and growing continuously.
- RDBMS can cope with large class-libraries: Multiple names for features and classes are required to be stored in the class library, representing overloaded feature names and multiple vendor classes. Once the number of classes begins to reach thousands or tens of thousands, as is likely in large development environments, the size and complexity of the class library are such that more sophisticated tools like database systems will be required.

Chapter 3

Design of the Class Library Management System(CLIMS)

In this chapter, the design of CLIMS is proposed. The conceptual view of the overall system is in figure 5. The approach we chosen for the management of the class-library is to put CLIMS as a front-end on the class library files and to let reusers access the classes efficiently through CLIMS. The design procedure can be summarised as follows:

- Identify the *reuse information* of a class.
- Transform the reuse information of a class into a conceptual structure of relational database management systems.
- Design the structure of CLIMS.

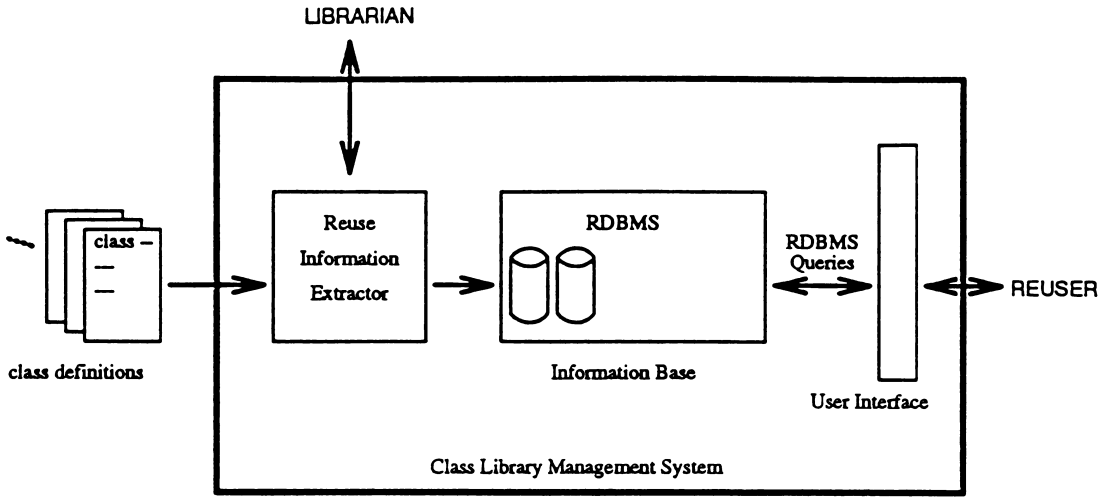


Figure 5: Overview of CLIMS

3.1 Systems Requirements

The Class-LIbrary-Management-System(CLIMS) has been designed in order to facilitate reuse of software by centralised management of C++ classes in distributed software development environments. The technical requirements for CLIMS are:

- mechanism to support locating and understanding classes in the system,
- classification scheme whose organisation is based on the attributes that characterize the requirements of software developers,
- mechanism for security control and consistency check, and
- user-friendly interface.

As a non-technical requirement, cost-effectiveness of system development is important to provide the software community with a motive that promotes the reuse

environment.[1]

3.2 Preliminary Discussion

3.2.1 Reuse Information about a Class

The development process of object-oriented application is connecting objects to each other and providing them with parameters [WBJ90]. However, finding classes for the objects, understanding each class and finally inter-connecting the classes for a new application are not easily done unless there exists some useful information called *reuse information.* The reuse information includes:

1. general descriptive information about a class,
2. environmental information about a class,
3. information about the relationship which a class has with other classes, and
4. information about members[2] of a class.

Figure 6 shows the necessary reuse information of a class in C++, which have been chosen for our system.

[1]The need for initial investment is one of obstacles against reuse.
[2]members in C++ consists of member functions and member attributes

Reuse Inforamtion for a C++ Class

General Information	
class-name	*identifier*
category	*application domain of the class*
keywords	*list of keywords for the class*
description	*brief description of the class*
version	*version of the class*
class-type	template /abstract / normal
template parameters	the parameters of the template class / empty
Relationship Information	
uses	*list of class names that are used in the class*
super-classes	*list of super-class names*
Member Information	
member-attributes	*list of member-attribute names*
member-functions	*list of member-function names*
Environmental Information	
machine	*machine name where the class is developed*
compiler	*compiler where the class is developed*
files	*the list of file names that should be linked together to excute the class*
For each super-class	
class-name	*name of the super-class*
access-specifier	public / private
For each member-attribute	
attribute-name	*name of the attribute*
attribute-type	*type of the attribute*
description	*brief description of the attribute*
access-specifier	private / protected / public
For each member-function	
function-name	*name of the function*
return-type	*return type of the function*
arguments	*the list of arguments and their types*
access-specifier	private / protected / public
pre-condition	*pre-condition of the function*
post-condition	*post-condition of the function*
description	*brief description of the functioon*

Figure 6: Reuse Information of a C++ Class

3.2.2 Classification Scheme of CLIMS

The basic principle of classification is gathering classes into groups such that the members of the same group share common properties [MBK91]. This grouping could help a user to locate the best candidates for reuse more effectively. Also, a classification scheme in a class library management system has much to do with the efficiency of retrieval: if the classes in the library are organised by the attributes that characterize the requirements of software developers, the probability of retrieving relevant components increases [RDF87]. So the reuse information which is mentioned in subsection 3.2.1 is the most desirable attribute for a classification code.

Prieto-Diaz has criticised *enumerative* (hierarchic) classification for reusable software components and suggested what is called a *faceted classification* scheme where each reusable software component is described by a component descriptor consisting of six classification facets [RDF87]: functionality of the software component; object manipulated by the program; a medium that serves as a locale where the function is executed; system-type; functional area; and setting.

The classification scheme for CLIMS is a mixture of the faceted and enumerative schemes. We adjusted the *faceted classification* scheme since the software components we are concerned about are not normal programs but the classes in OOP: the class is usually built around an *object* rather than the *functionality* from which the faceted scheme is built around. In most cases, the class name represents an *object*: some linguistic variation can be solved using a wildcard character (eg. sort* can cover sort, sorting, or sorts) or a system thesaurus.

The major classification mechanisms of CLIMS are the following:

- Assignment of a category for each class: the category specifies the application domain of the class.
- Assignment of keywords for each class: the keywords specify the representative words for the class.

Basically every attribute (one attribute or a combination of attributes) for a class can be used like an index for retrieval since we used a relational database table to represent the attributes of the class: the query languages for RDBMS allow a retrieval operation with any attributes; eg. a function name (or its linguistic variants) can be used for locating the class(es) that contain(s) the function as a member.

3.2.3 Reuse Information Display:Ranked List and Flatform

When a user sends a query to find a reusable class, CLIMS displays every class that matches the user's query partially as well as fully. Allowing retrieval of partially matched classes increases the so called *recall* of the system. Recall measures the proportion of relevant material that is actually retrieved. If the size of a class library is large, the number of candidate classes that are relevant to user queries will be also large enough to embarrass reusers. To help user's selection among the candidate classes, CLIMS displays the names of the candidate classes ordered by the degree of matchability between user requirements and reuse information

for classes. Then the user may try to see the reuse information about the classes in the list one-by-one, top-to-bottom, until he finds the closest one to his requirements.

To ensure if a class is reusable for a new application, the user should understand the functionality of the class. If the user wants to know the functionality of a class and selects a menu for it, CLIMS will display the protocol of the class. But, if the class of which the user wants to see the protocol, is a *derived class* (in C++ terminology), capturing the protocol of the class is not so simple. The inheritance technique that facilitates the reusability, ironically, makes the understanding of a code complicated. For example, assume the case where a class *Circle* is inherited from a class *Shape*. Figure 3.2.3 shows the class definitions of the two classes in C++. Inheritance mechanism facilitates the reusability of code through a subclassing technique: a new class(derived class) can be built based on an existing class(base class) just by defining the difference between the new class and the existing class. In figure 3.2.3, the class *Circle* redefines the definition of function, *draw*, in the base class and add two new functions, *Expand* and *Contract*. Here, the protocol of the class *Circle* consists of

- the four member functions of the class *Circle* : Circle, draw, Expand, and Contract.
- the member functions of class *Shape* : Shape, orgx, orgy, move.

And so, the user who wants to find the protocol of a class, say 'D', should also check the protocol of its base class(es). If the base classes are again inherited from another class, say BB, the protocol of the class 'BB' should be also checked

```
class Shape {
   Point org;  //private variable for origin
public:
   Shape(const Point& p) : org(p){}; //public ft for
                                       initializing origin point
   int orgx() const;  //public ft for returning
                        the x-coordinate of origin
   int orgy() const;  //public ft for returning
                        the y-coordinate of origin
   virtual void move(int dx, int dy);  //public ft
                                         for moving the origin
   virtual void draw() const = 0; //public ft : pure virtual ft
};

class Circle : public Shape { // derived from the class Shape
   int rad;           //private variable for radius of circle
public:
   Circle(const Point& c, int r) :
                          Shape(c); //public ft
                                      for initializing the circle
   virtual void draw() const; //public ft for drawing the circle
   void Expand(int ExpandBy); //public ft for expanding the circle
   void Contract(int ContractBy); //public ft for contracting the
                                    circle
};
```

Figure 7: Example of class definitions

to decide the protocol of the class 'D'. So, understanding functionality of a class entails navigating along the inheritance hierarchy. To help a user avoid this complicated as well as tedious job, the *controller* of CLIMS visits the ancestor classes along the inheritance hierarchy using the inheritance relationships stored in the DB storage instead of a user, and displays a proper protocol : this protocol is much like so called a *flat* form in Eiffel terminology. This ability of calculating the protocol can also remove the need for local browsing to some degree. This technique of the interface display will be necessary for a user to understand the functionality of a class especially when the class hierarchy is *deep*. Another advantage of this technique is that it can partly solve the maintenance problem, where understanding the functionality of a software component is essential. The major algorithm for calculating the protocol of a given class will be discussed in chapter 4.

3.3 Design of Data Structure for CLIMS

In this section, the relational dictionary approach for the design of data structure in CLIMS is proposed. The Entity-Relationship(E-R) data model is used to represent the reuse-information in C++ class definitions. Based on the E-R data model, the logical data structure of relational database is produced.

3.3.1 Conceptual Design Using E-R Data Modelling Technique

The Entity-Relationship (E-R) data model [P.P76] is a semantic data model that is based on a perception of a real world which consists of a set of basic objects called *entities* and *relationships* among theses objects. The main purpose of E-R data-model is to facilitate a database design by allowing the specification of an *enterprise scheme* which represents the overall logical structure of the database.

The model uses two distinct modelling constructs, *entity* and *relationship*, to grasp the real world which comprises a set of basic objects (corresponding to entity) and relationships among these objects. There are two types of entities: *strong* and *weak* entities. A strong entity is an object that has an independent existence. On the other hand, a weak entity is an object, the existence of which is dependent on the existence of other associated entities. Entities may have attributes, each of which describes a piece of information of the objects.

A relationship describes an association among entities or a (relationship-)entity that serves to interconnect two or more other objects. A relationship can be either *total* or *partial*. If every instance of entity E participates in at least one instance of relationship R, then the relationship of E in R is said to be *total*, otherwise it is *partial*. An E-R relationship can be one-to-one, one-to-many, or many-to-many. Like an entity, a relationship can be strong or weak depending on the types of the entities participating in the relationship. If a participant of the relationship is a weak entity, the relationship tends to be a weak relationship.

A relationship may also have its own attributes. In addition, a real world relationship can be represented in two distinct ways in E-R representation: that is, by an attribute or by a relationship. If the attribute is printable (i.e., literal), it is directly represented by an attribute, otherwise it is represented by an explicit relationship.

One of the distinct features of the E-R model is the presence of an E-R diagram. The following is a summary of notation:

- A rectangle represents an entity set; A doubly outlined rectangle denotes the weak entity.
- An ellipse represents an attribute.
- A diamond represents a relationship set.
- A line links an attribute to an entity set or an entity set to a relationship set.

E-R Modelling of Reuse Information in a C++ Class

The E-R data modelling technique is used in the design of CLIMS in order to represent the reuse information in class definitions in the form of a logical structure of a relational database (conceptual schema of RDBMS). This schema is again used for creating base tables, which RDBMS stores data in and does operations on. Figure 8 is the E-R diagram which represents the reuse information (see figure 6) in C++ class definitions.

Expected page number is blank in the original print copy.

The entity set **class** is itself associated with the relationship sets, **uses** and **inherits**. The **uses** means the case when a class uses the definition of another class for its type definition: i.e. the construction with respect to the way of reuse. The entity set **class** has three weak entities. A class comprises a set of member functions and member attributes, which are represented by the entity sets, **MemFunc** and **MemAttr**, respectively. Also, a class is allowed to have as many keywords as it can. This set of keywords for each class is represented by the **keywords** entity set. All the bubbles connected to the class entity represent attributes of the class entity. The **category** is the category (application domain) of a class, which is to be used to classify classes according to their application domain. The **class_type** tells whether a class is generic (template in C++ terminology) or not. The **file** contains names of the files that should be linked together to execute a class. Other attributes are self-explanatory.

The entity set **MemFunc** has a weak entity, **arguments**. The **arguments** embodies information about arguments of a member function and their types. The attribute **description** means brief descriptive information about each function. All the bubbles linked to the entity set **MemFunc** are attributes of a member function. The **precondition** and **postcondition** of each function formally specify the pre- and post-condition of each function, the concept of which is borrowed from the Eiffel language which supports the *assertion* with a language structure. The **f_type** denotes whether a function is pure-virtual-, constant- or normal-function. The attribute **argu_type_list** is a set of types of arguments for a function. By using the concept of overloading, C++ allows the same name for different functions as long as the number and types of arguments are different.

So, we have the attributes **argu_type_list** to distinguish a function from others that share the same name.

The entities **MemAttr** and **keywords** are self-explanatory.

3.3.2 Logical Design for Relational Schema

Transforming E-R Diagram to Relational Database Schema

Once the E-R diagram is obtained, it has to be transformed into a logical data structure of designated database systems. Since RDBMS is employed as our choice of a database system, the rules for transforming the E-R diagram into a relational database schema are needed. The following are summarised rules [KS91]:

- Represent a strong entity set, E, with descriptive attributes $a_1, a_2, \cdots, a_n$ by a table called E with n distinct columns each of which corresponds to one of the attributes of E.
- Let A be a weak entity set with attributes $a_1, a_2, \cdots, a_r$. Let B be a strong entity set on which A is dependent. Let the primary key of B consists of attributes $b_1, b_2, \cdots, b_s$. Represent the entity set A by a table called A with one column for each attributes of the set: $\{a_1, a_2, \cdots, a_r\} \cup \{b_1, b_2, \cdots, b_s\}$
- Let R be a relationship set involving entity sets $E_1, E_2, \cdots, E_m$. Let the primary keys of entity sets $E_1, E_2, \cdots, E_m$ are $\{b_{11}, b_{12}, \cdots, b_{1s}\}$, $\{b_{21}, b_{22}, \cdots, b_{2t}\}, \cdots$, $\{b_{m1}, b_{m2}, \cdots, b_{ms}\}$, respectively. Let the attributes of the relationship R itself be $a_1, a_2, \cdots, a_r$. Represent the relationship by a table called R with

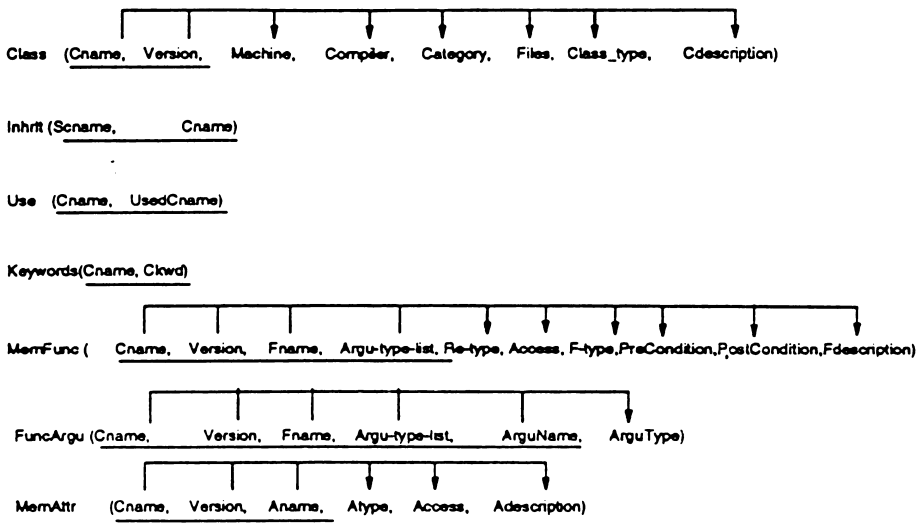


Figure 9: Relational schema and its functional dependency

n distinct columns each of which corresponds to one of the attributes of the set: $\{b_{11}, b_{12}, \cdots, b_{1s}\} \cup \{b_{21}, b_{22}, \cdots, b_{2t}\} \cup \cdots \cup \{b_{m1}, b_{m2}, \cdots, b_{ms}\} \cup \{a_1, a_2, \cdots, a_r\}$.

Relational Schema for Class Definition

Apply the rules (for translating an E-R diagram to the relational schema) to the diagram in figure 8 and get the relational schema. The schema produced are in figure 9. The fields underlined represent a key for a relation. The directed lines over the relations depict the functional dependencies between fields of each relation : Given a relation R, attribute Y of R is *functionally dependent* on attribute X of R; in symbols, R.X –> R.Y [Dat90].

One important goal in a database design is the concept that each fact should be stored in one place only. Redundancy may cause problems [Dat90]: so called *anomaly* in manipulation, inconsistency, incorrectness of database and storage waste. The *normalisation* theory has evolved for achieving this concept: "one

fact in one place". There exist many different properties, or "normal forms" for relation schemes with *functional dependencies* [KS91]:

- A relation is in the *first normal form* (1NF) if and only if all underlying simple attributes contain atomic values only.
- A relation is in the *second normal form* (2NF) if and only if it is in 1NF and every non-key attribute is fully dependent on the primary key.
- A relation is in the *third normal form* (3NF) if and only if it is in 2NF and every non-key attribute is non-transitively dependent on the primary key.

The functional dependencies of the relations shown in figure 9 illustrate that the relations produced from the E-R diagram are at least in 3NF.

3.4 Design of Functions for CLIMS

3.4.1 Logical Design of CLIMS

A data flow diagram(DFD) [Dav83] is used to design a logical model of CLIMS. This logical model does not depend on hardware, software nor data structure of file organisation. DFD uses four symbols to form a picture of a logical system. A square defines a *source* or *destination* of data. A rectangle with rounded corners(or a circle) represents a *process* that transforms data. An open-ended rectangle is a *data store*. An arrow is used to identify a *data flow*.

DFD is used as a starting point for the system design in order to portray the

system in terms of its component pieces, with all interfaces among the components indicated [Dem79].

A high-level DFD for CLIMS is prepared (see figure 10). Process 1 is concerned with maintenance; while certainly important, this is a support function. Similar arguments might be advanced for processes 4 and 5; all involve support functions that essentially summarise or report the results of the primary processes, *extractor*(process 2) and *control system*(process 3). *Extractor* collects reuse information about classes and stores the reuse information into a class library. The reuse information consists of the information which is extracted from the h-files and which is obtained from the system manager.

The *control system* is responsible for performing the following functions:

- Verify user identity and queries.
- Transform user queries into database queries.
- Retrieve reuse information from the class-library.
- Format the retrieved reuse information.

A logical model of the subprocesses involved in *control system* is shown in figure 11.

Note that the design assumptions do not imply a physical implementation. The class-library, security-, and user-information are shown as separate stores because the data are logically different, not because they must be stored on physically separate files or devices.

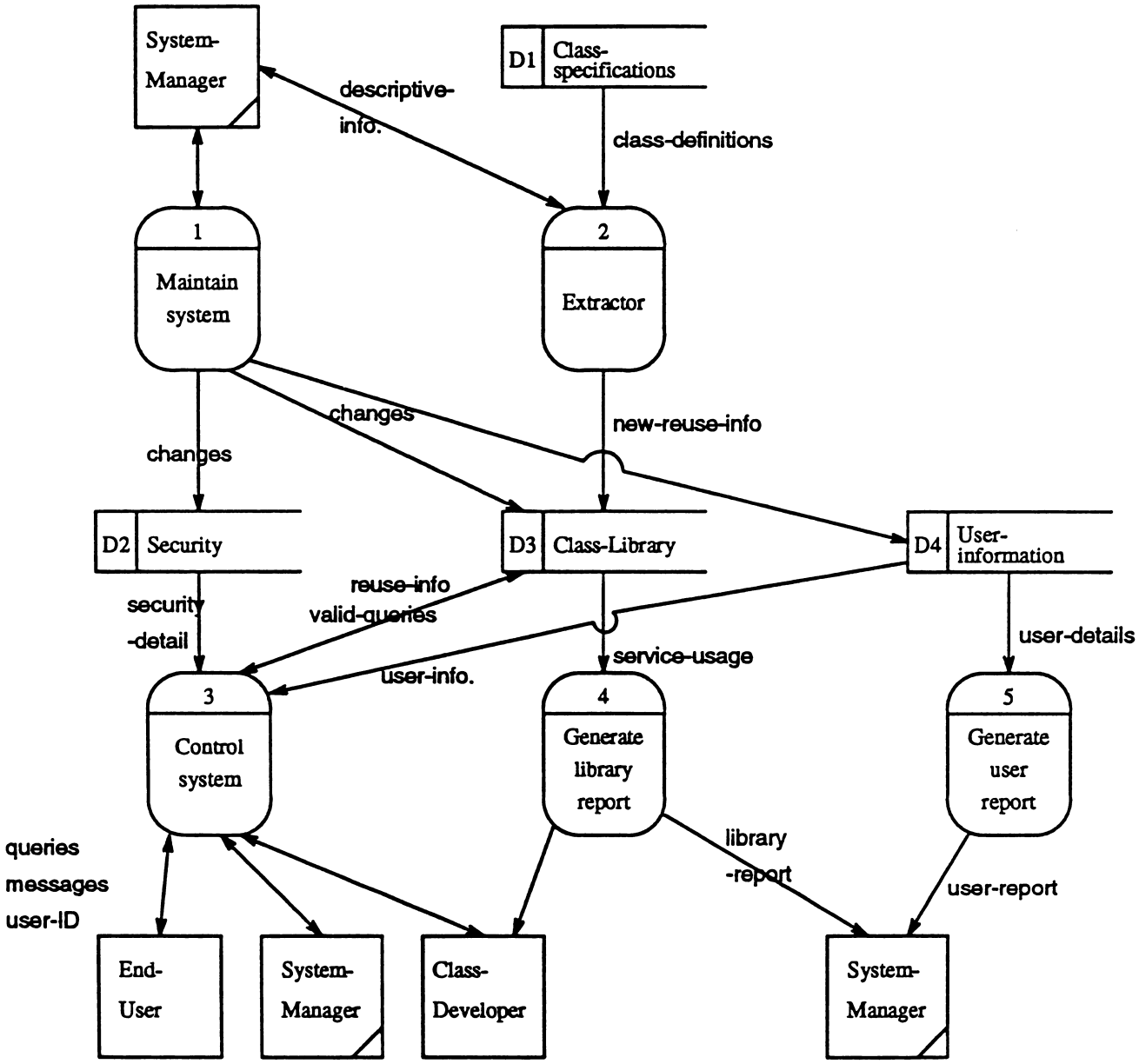


Figure 10: High-level data flow diagram of the system

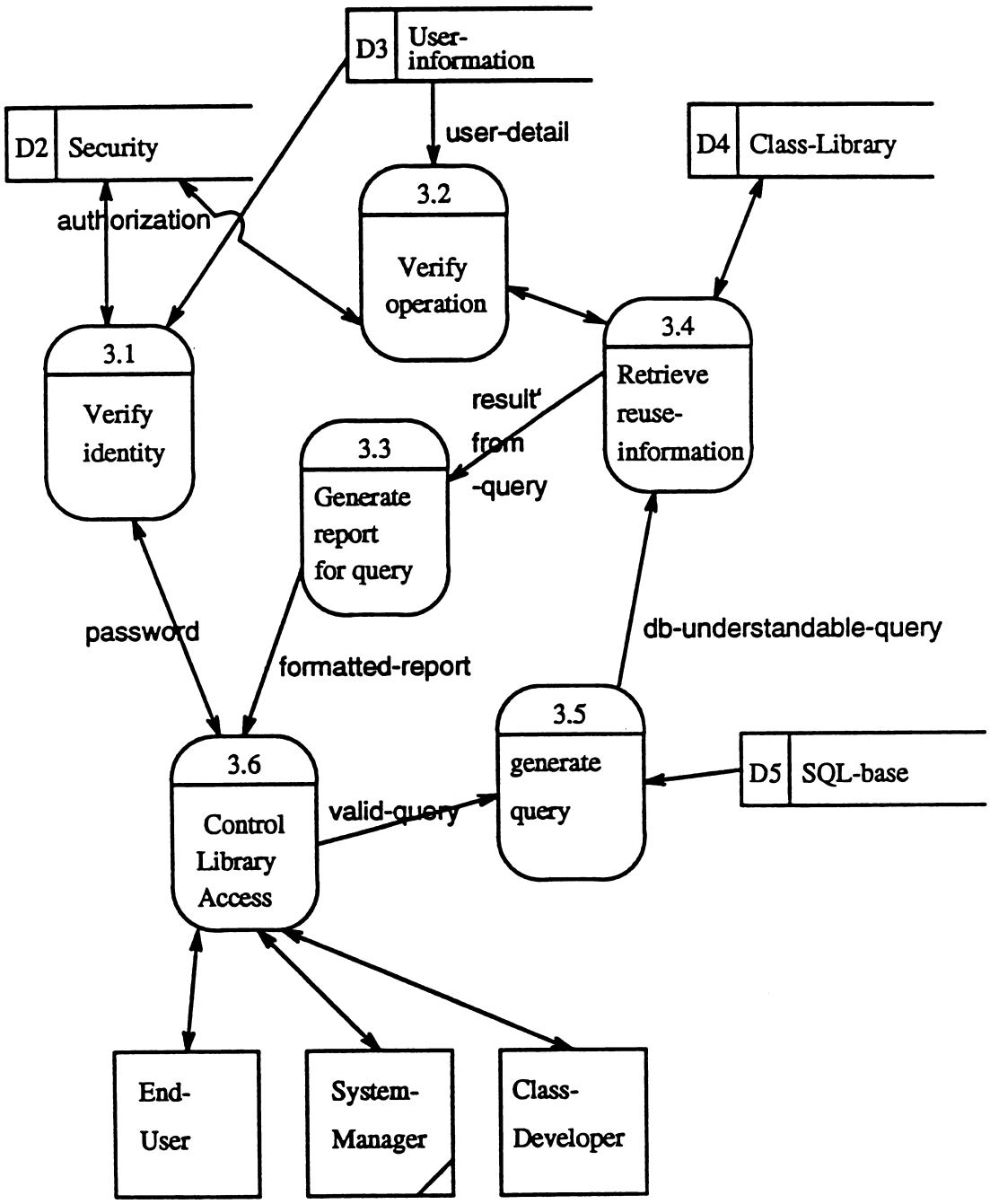


Figure 11: An explosion of the system control process

3.4.2 Physical Design of CLIMS

A system flow chart [Dav83] is one of tools for describing a physical system. The basic idea of the system flow chart is to provide a symbol to represent, at black-box level, each discrete component in the system-programs, files, forms, procedures, and so on. The symbols for the system flow chart allow the analyst to represent the actual device or processes that make up the system.

The system flow chart for CLIMS which is converted from the logical model of CLIMS (see figure 10) is shown in figure 12. All the data stores and parts of programs, *System control*, *Management report generator* and *Customer report generator*, inside the dotted-boundary will eventually be implemented by RDBMS. The tasks of 'Maintain system' in the DFD (see figure 10) are also treated as a special set of routines within the RDBMS; these functions are accessed through a maintenance terminal.

The two programs, 'system control' and 'extractor' are major parts of CLIMS which have to be implemented by us.

3.4.3 Detailed Design of CLIMS

The *hierarchy chart*[Dav83] is a tool for representing a top-down structure of a program. A complete implementation plan for each program which is identified during the system design (see figure 12) is specified one-by-one by use of the hierarchy-chart. The system flow chart in figure 12 identifies four different programs, five files or libraries, and several reports. As already referred to, the

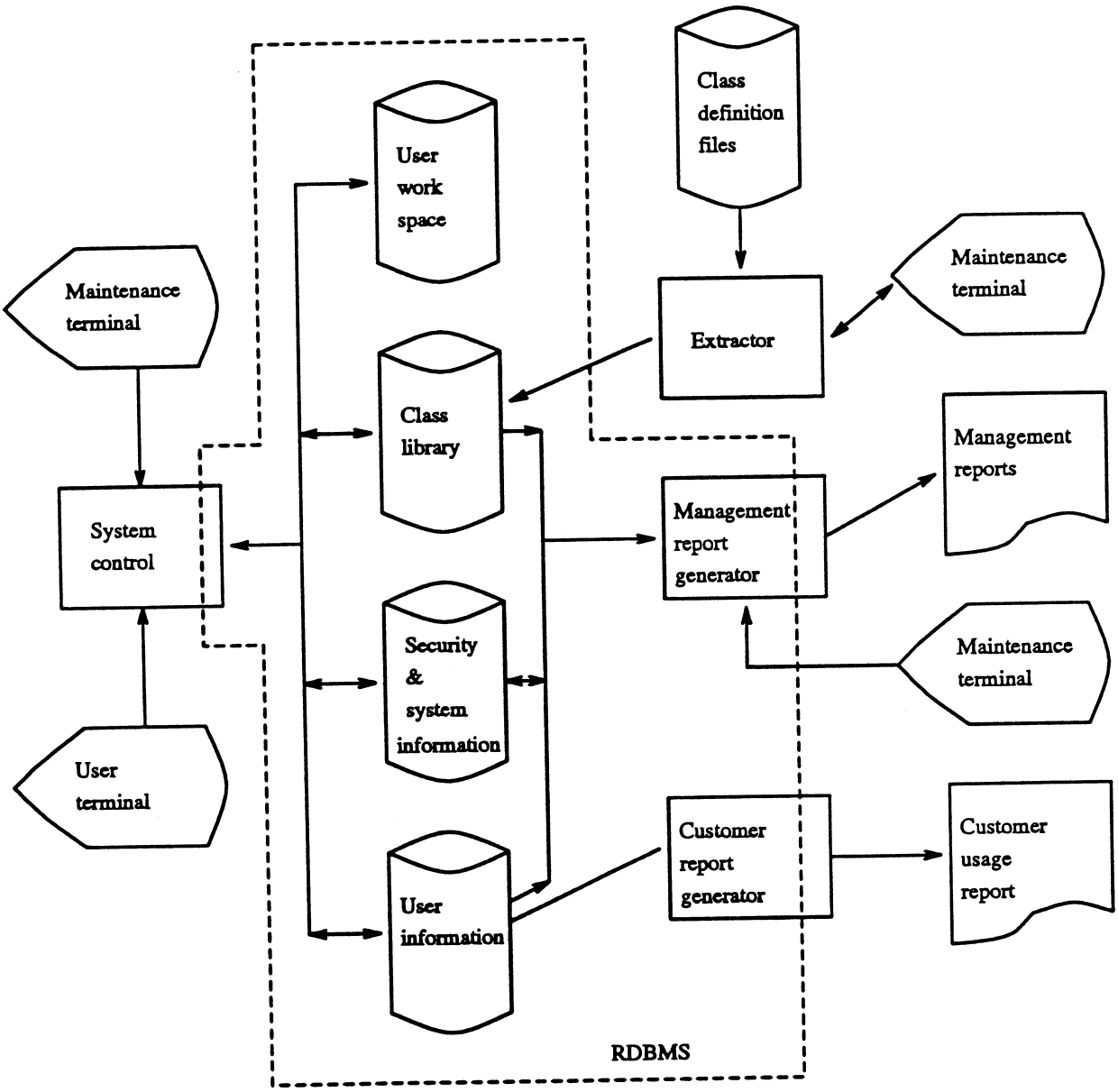
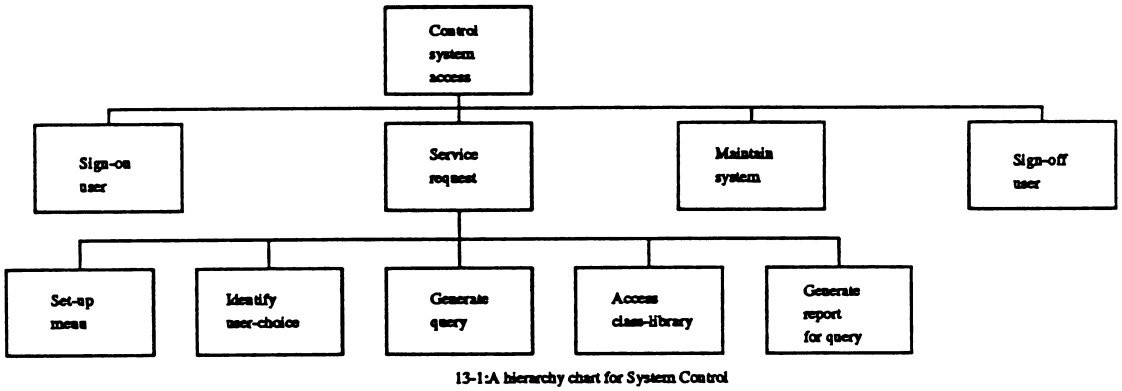
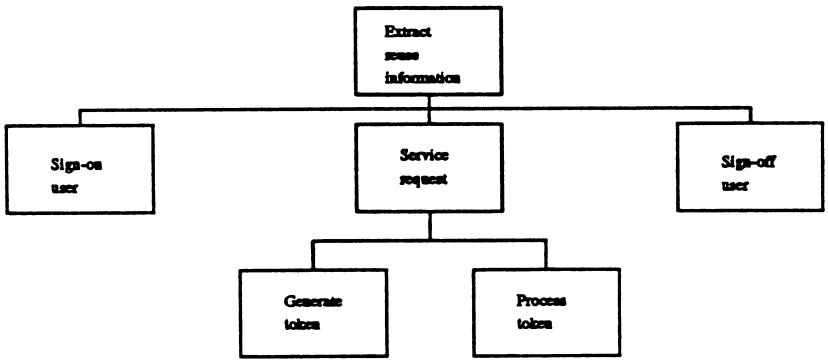


Figure 12: A flowchart of system design



13-1:A hierarchy chart for System Control



13-2:A hierarchy chart for Extractor

Figure 13: A hierarchy chart of major programs

existing RDBMS could implement the two programs and four data stores: Management report generator, Customer report generator, User work space, Class-library, Security & system information, and User information. Just by providing a set of database commands, the RDBMS could be tailored for our use.[3]

To describe the planning for *system control* and *extractor*, we decide the functions that will be included in the two programs and propose the hierarchical organisation of the functions in each program using *hierarchy chart*. The program hierarchy charts for the two programs are in figure 13. The function *Sign-on user* is for verifying a user for security purposes: for different kinds of users, different kinds of operations will be permitted on different parts of the database. The function *Maintain system* is for general maintenance problems for the system.

The *service request* is a core function in the *system control* which sets up menus for users, identifies a users' choice of a menu, generates database-understandable queries, accesses the class-library, and displays the retrieved information in an easy-to-understand forms.

The *service request* in *extractor* consists of two major functions: a token-generator and a token-processor. The *token-generator* reads through the h-file and generates a sequence of tokens for the token-processor. The *token-processor* processes the tokens in order to: extract reuse information; check the syntax and save the reuse information into the class-library.

[3]The implementation detail will be discussed in the chapter4.

3.5 Summary

The design detail of CLIMS has been proposed in this chapter.

First, the system requirements for CLIMS have been specified in order to guide the design procedure.

The reuse-information of a C++ class has been defined, which is core information to be managed by CLIMS. The classification is a grouping technique which affects the efficiency of retrieval manipulation. The mixture of the enumerative- and faceted-classification scheme has been employed for CLIMS. Two important ways of displaying information d have been discussed. The ranked list is essential to help reusers with locating the most proper classes efficiently, especially when the number of classes which match user requirements fully or partially is large. The flat-form is a concept taken from the Eiffel language in order to relieve reusers of the complicated navigation along the hierarchy of classes. This technique allows reusers to understand the functionality of classes efficiently.

The design of the data structure for CLIMS has been discussed. Only the reuse-information rather than the source code of C++ classes is managed in CLIMS. Since with respect to reusers' view of classes, the reuse information is all the information required. This approach also strengthens the level of abstraction by providing physical encapsulation[4]. The reuse information of C++ classes is represented in the form of a set of tables inside RDBMS to facilitate efficient

[4]In a source code level, for example, the private members are physically visible though they are invisible logically

management. The relational dictionary technique is discussed to show how to transform the reuse information into the table-form of relational schema using the E-R data modelling technique.

Lastly, the design of functions in CLIMS is proposed in detail. The logical model of CLIMS is constructed using DFD. The system flow chart is developed based on the logical model to show the physical system organisation of CLIMS. The complete implementation plan of major programs in the system flow chart is proposed using a hierarchy chart.

Chapter 4

Implementation of the CLIMS

4.1 Overview of Implementation

The prototype of CLIMS has been developed by using an embedded-C [emb89a][emb89b] with the INGRES(version 6.2) system under a UNIX environment. The comprehensive view of the implementation is shown in figure 14. The backend in figure 14 is the INGRES relational DBMS per se. It provides all the basic DBMS functions, including full support for everything in the database query language: data definition, data manipulation, data security and integrity, and so on. It also serves as a transaction manager, providing the necessary concurrency control and recovery support. The frontend consists of a set of built-in facilities[1] for assisting in the process of developing database applications and application programs. The two major application programs are *System Control* and *Extractor*, the implementation of which will be discussed in detail in this chapter.

[1]facilities include several distinct subsystems: INGRES/QUERY, INGRES/REPORT, INGRES/GRAPHICS, INGRES/FORMS and INGRES/APPLICATIONS

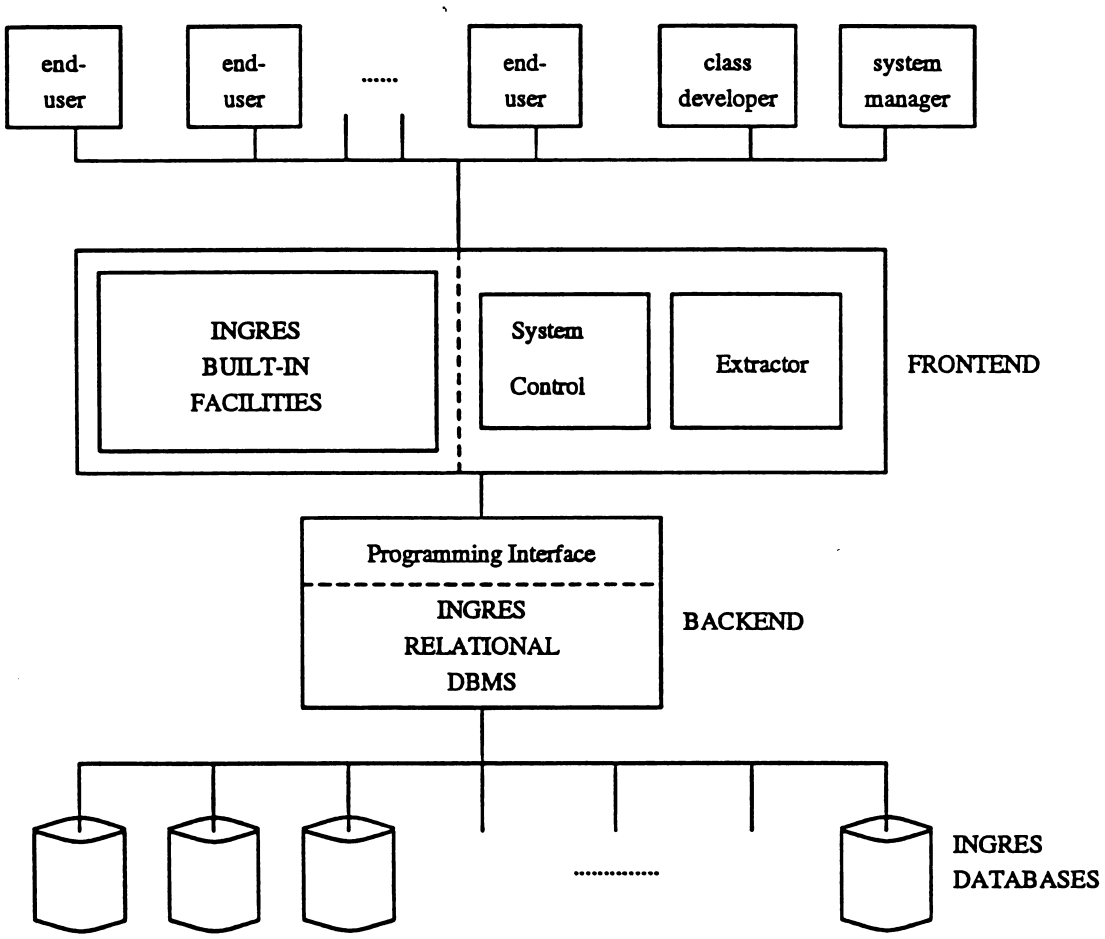


Figure 14: Implementation of CLIMS by utilising the INGRES system

4.2 Preparation of Databases and Security Control

Creating base tables inside INGRES is easily done by using query languages like SQL. INGRES also supports a high-level forms-based interface. Just by following the relations which we have proposed in section 3.3, tables for the reuse information about C++ classes has been prepared.

Depending on the kinds of users, CLIMS supports different views and allows different levels of operations. There are three kinds of users involved in CLIMS:

1. The class library administrator who is responsible for the management of CLIMS: he will be allowed to access all the tables inside CLIMS and to do all operations on the tables.

2. The software developer who writes codes for classes: he will be allowed to access all the tables inside CLIMS and to do restricted operations on the tables.

3. The end-user who uses the existing classes in the library for his own application: he will be allowed to access restricted parts of the tables and to do restricted operations on the tables.

These different levels of operations on different views of tables can be easily implemented by use of the INGRES facilities: *view* and *grant* command. 'View' is a virtual table[2] which can be thought of as different ways of looking at the real

[2]a table that does not directly exist in physical storage, but looks to the user as if it did

tables. Accordingly, we created three different views for members of a C++ class according to their access-specifiers such as the following:

```
CREATE VIEW public_function
AS SELECT Cname,Version,Fname,Argu_type_list,Re_type,F_type
FROM MemFunc
WHERE Access = 'public';

CREATE VIEW protected_function
AS SELECT Cname,Version,Fname,Argu_type_list,Re_type,F_type
FROM MemFunc
WHERE Access = 'protected';

CREATE VIEW private_function
AS SELECT Cname,Version,Fname,Argu_type_list,Re_type,F_type
FROM MemFunc
WHERE Access = 'private';
```

Through a similar procedure, different views of member attributes have been also created: public_, protected_, and private_attribute.

Using 'grant' command, different privileges of operations for different levels of users have been created as follows:

```
GRANT ALL ON  Class, Inhrlt, Use, Keywords, MemFunc,
              Assertion, FuncArgu,MemAttr
TO    library-administrator;
```

```
GRANT SELECT ON Class, Inhrlt, Use, Keywords,
                MemFunc, Assertion, FuncArgu, MemAttr
TO    software-developer;

GRANT SELECT ON Assertion, Class, FuncArgu, Inhrlt, Keywords,
                public_function, public_attribute, Use
TO    end-user;
```

Note, the end-user is allowed to access only the public members i.e. the views, public_function and public_attribute.

As seen above, the combination of two techniques, view and grant-command allows us to provide layered views which are flexible enough to deal with different levels of users efficiently. This layered views result in the following advantages:

- the user's perception is simplified,
- the encapsulation is supplemented,
- the security is provided for hidden data, and
- the reliability of library is improved by restricting operations.

4.3 Implementation of the Extractor

The C++ program system, as in C, conventionally consists of a set of files that contain class definitions, method implementations and global declarations. Accordingly, it is assumed that a class definition is in h-file and the implementation

of a class is in C-file. Since the target source of the extractor is only the class definition, the extractor takes the h-files as input data, processes the class definition of each class one-by-one to extract the reuse information, and stores the extracted reuse information from the h-files into the ready-made base tables in INGRES.

Processing of the class definition is much like the function of a *parser*. So, developing the extractor involves the technique of developing a *lexical analyser* and a *syntax analyser* in a compiler. The development steps for the extractor consist of three phases:

- Get the BNF form of a C++ class definition.
- Develop a *token-generator*: the token-generator scans the stream of characters in the h-file and returns tokens that are a sequence of characters having a collective meaning.
- Develop a *token-processor*: the token-processor gets the tokens from the token-generator and groups them into grammatical phrases to check the syntax of the class definitions. While in the process of syntax checking, the token-processor saves the necessary information (eg. class name, function name, etc.) in temporary storage. After finishing the process of each class, the temporarily stored information is actually saved in the INGRES base tables using embedded-SQL.

Figure 15 illustrates the interaction among the processors inside the extractor. The 'token-generator' and 'token-processor' have been implemented by using the

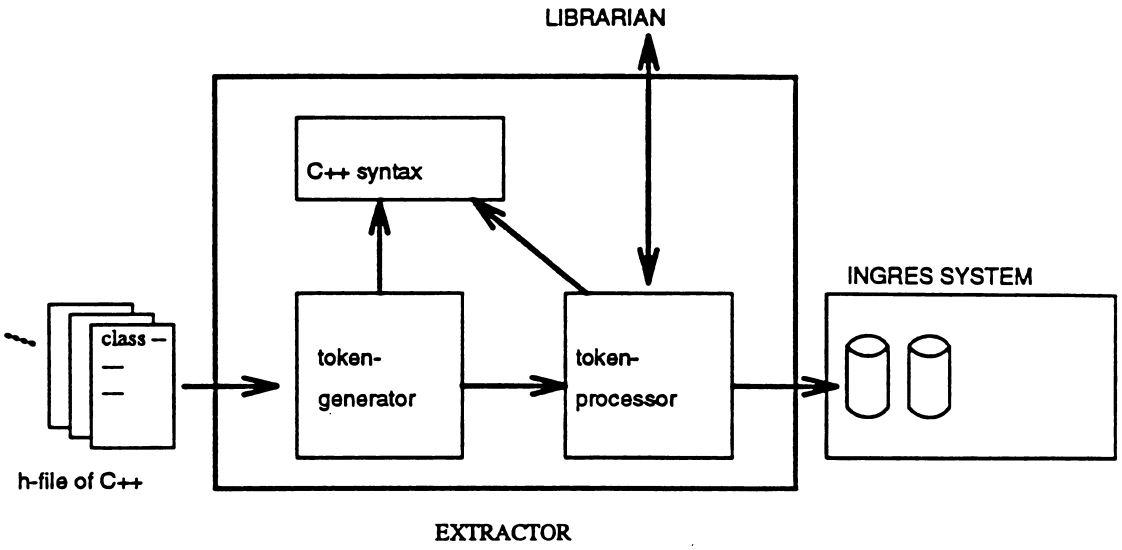


Figure 15: Structure of extractor

Lex and *Yacc* [MB90]. Lex and Yacc are tools that facilitate creation of C-routines that analyse and interpret an input stream according to the given grammars. Lex accepts a specification file which includes regular expressions for pattern matching as input and produces a C-routine which performs a lexical analyser as an output. This routine reads a stream of characters and generates a sequence of tokens. Yacc accepts a specification file which codifies the grammar of a language as an input and outputs a parsing routine. This routine groups tokens from the lexical-analyser into meaningful sequences and invokes action routines to act upon them.

The existing specification files [Ros90] for Lex and Yacc[3] are borrowed and adjusted for our purpose by inserting action-routines we want inside the specification file.

A part of C++ syntax specification with action-routine inserted in it is illustrated

[3]They are for generating a lexical analyser and parser for a C++ compiler

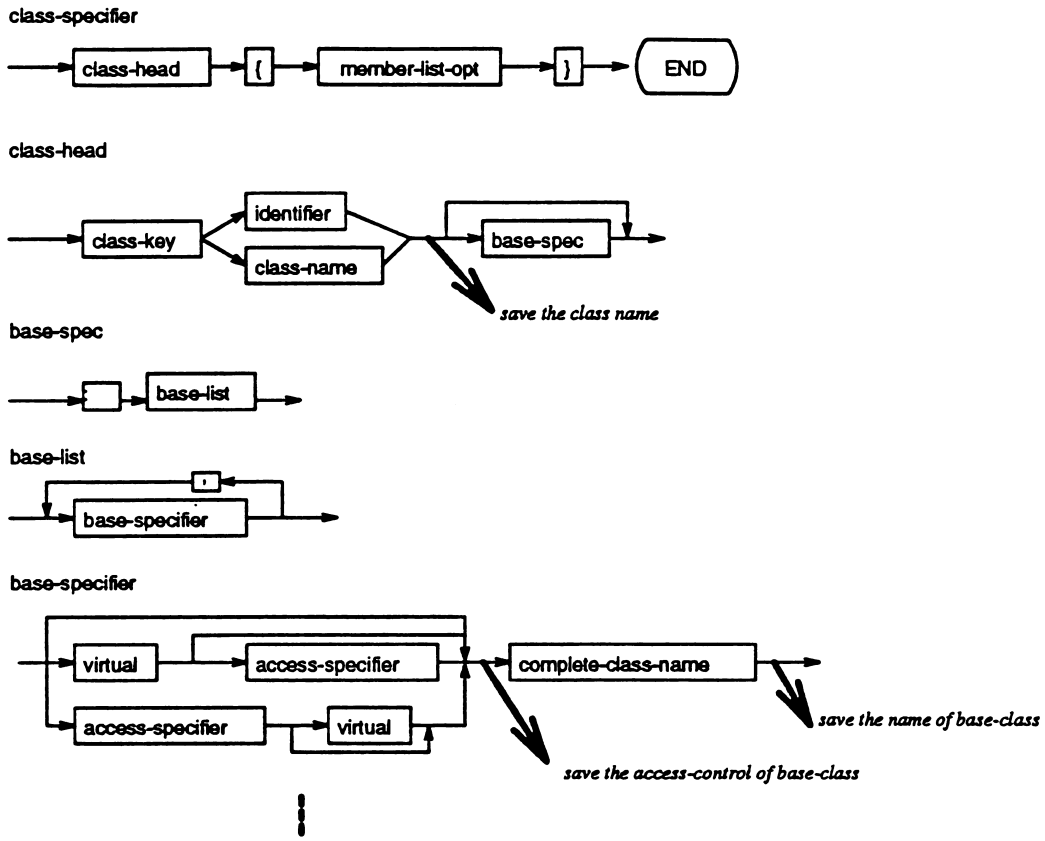


Figure 16: Syntax diagram and action routines

by syntax diagrams in figure 16. The directed thick lines denote the action-routines inside the token-processor. For example, after processing *class-head*, the token-processor saves the name of the class being processed in temporary storage (see the second diagram in figure 16).

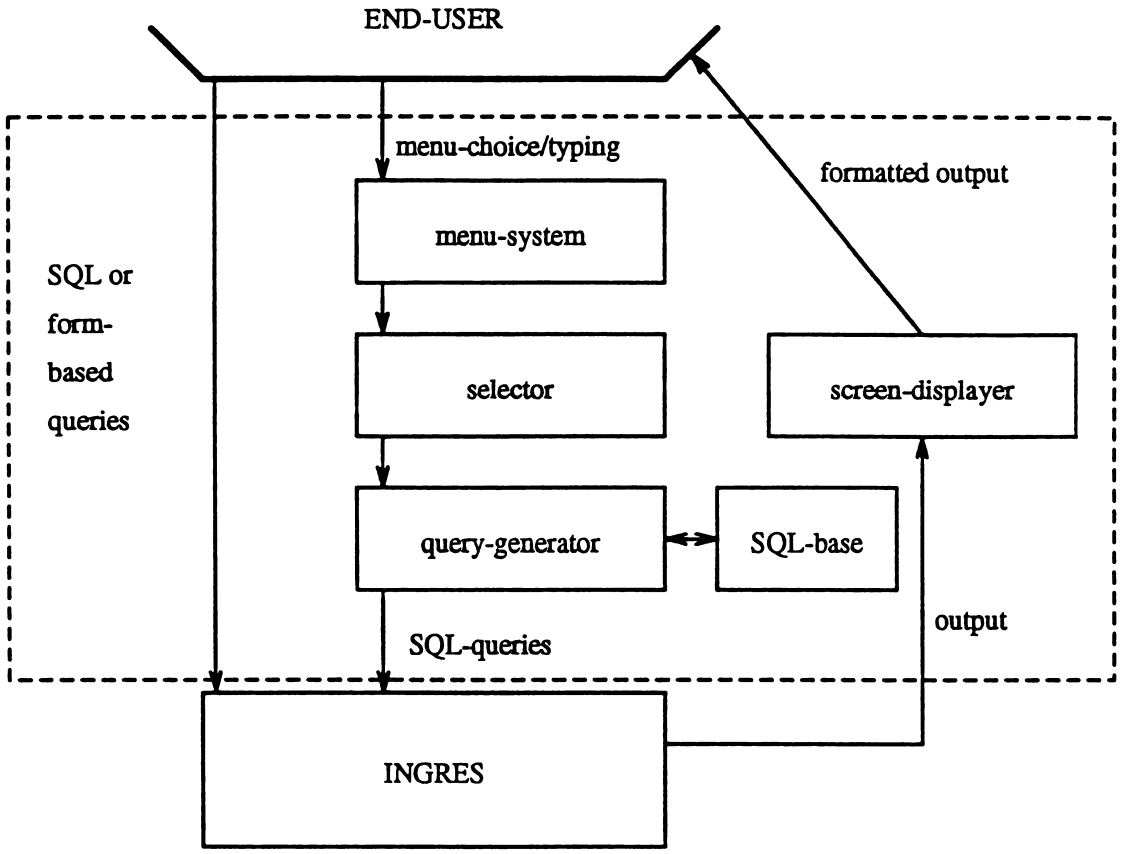


Figure 17: Overall structure of sub-functions of 'service-request'

4.4 Implementation of the Control System

In this section, the implementation detail of the *control system* will be discussed. Among the functions which are defined in the design phase (see figure 13), the function 'service request', which is a core part of the control system, is a major part we have concentrated on to implement the control system. Other functions like 'sign-on/off user' and 'maintain system' can be easily implemented by the use of the INGRES built-in facilities.

The overall interactions between sub-functions of the 'service-request' are shown in figure 17. A user accesses CLIMS usually through a menu front-end set up by

the *menu-system*. By selecting the menu, or typing-in responses according to the prompt from the menu-system, the user sends queries to the processor *selector*[4]. According to the queries, the selector chooses a procedure in the *query-generator*[5]. The procedure in the query-generator constructs a query by combining the SQL-queries in the *SQL base*, and forwards the query to INGRES. The output from INGRES is sent to the *screen-displayer*[6], which is responsible for restructuring the output into an easy-to-understand format.

The menu-system will simply display the list of options which a user can choose according to its sessions. The selector which maps the user's choice of menu to the corresponding procedure in the query-generator can be simply implemented by using a multi-way conditional branch statement like the *switch* statement in C. The screen-displayer can be implemented easily by using the INGRES facilities, INGRES/REPORTS. Figure 18 shows a sample session of user interaction with the control system.

4.4.1 Query-Generator

The *Query-generator* is responsible for generating queries for accessing INGRES according to users' choice of menu. A collection of procedures in embedded-C is prepared (named *SQL-base*) in order to make it easier for the query-generator to construct queries. Each procedure in the SQL-base implements a basic function which we have defined. The list of the basic functions is in figure 19. A simplified

[4]This corresponds to the function 'Identify user-choice' in figure 13
[5]This corresponds to the function 'generate query' in figure 13
[6]This corresponds to the function 'generate report for query' in figure 13

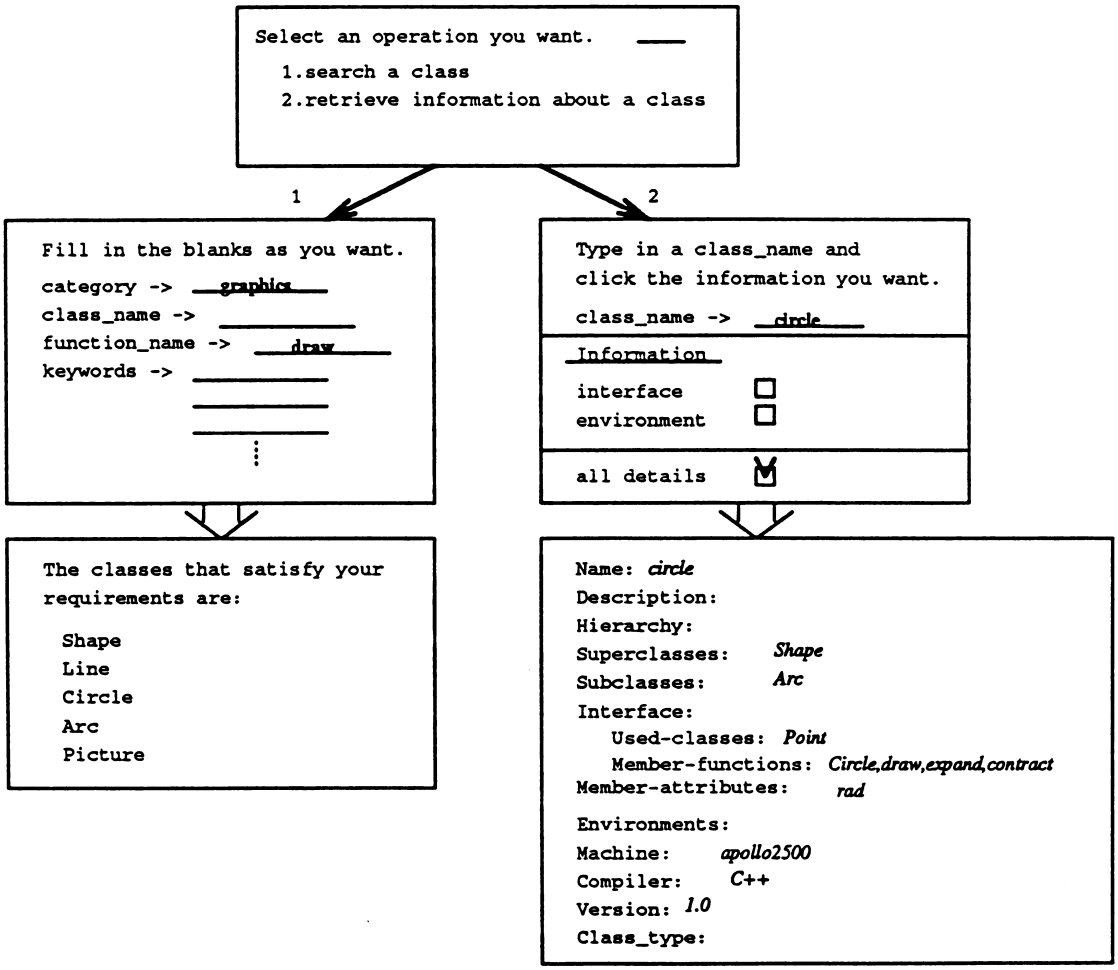


Figure 18: A brief description of user session

form of the implementation of the basic function $f1$ in embedded-C is shown in figure 20. The procedure $f1$ takes a class_name and version as input and produces the public member functions of the class as output. The output is stored in the temporary table named $f1\_result\_table$. The complete implementation of the procedures for the basic_functions is in appendix ??.

The query-generator is again composed of a set of procedures in embedded-C, each of which implements an expected user-query. Each procedure in the query-generator implements a user-query by invoking a sequence of procedures in SQL-base according to the user-query. For example, if a user query is "Find all public members of a specific class.", the corresponding procedure in the query-generator will invoke the procedures for the functions, $f1$ and $f2$ in the SQL-base (see figure 19), sequentially, which results in two tables, $f1\_result\_table$ and $f2\_result\_table$. The display of the two tables with a certain format will be what the user expected. For the other example, if a user query is "Find all the ancestor classes of a specific class.", the corresponding procedure in the query-generator sends the procedure $F8$, which implemented the function $f8$ in the SQL-base (see figure 19), to INGRES and saves the result from INGRES in a temporary storage, and recursively sends the procedure $F8$ to INGRES with the classes in the temporary storage as input. Repeating the above process until no more base classes are available produces complete answers.

The technique of preparing the basic units and combining them when it is needed achieves more flexibility than that of preparing a complete form of SQL-queries for every expected user query. Another advantage is that the range of user queries

```
-------------------------------------------------------------------------
f1:  class-name, version -> public member functions
f2:  class-name, version -> public member attributes
f3:  class-name, version -> protected member functions
f4:  class-name, version -> protected member attributes
f5:  class-name, version -> private member functions
f6:  class-name, version -> private member attributes
f7:  class-name, version -> class-type, files, machine, compiler,
                            category, description
f8:  class-name, version -> base class(es)
f9:  class-name, version -> sub-class(es)
f10: class-name, version -> used class(es)
f11: category -> list of classes in the given category
f12: keyword1 -> class name(s)
f13: keyword1, keyword2 -> class name(s)
f14: keyword1, keyword2, keyword3 -> class name(s)
f15: keyword1, keyword2, keyword3, keyword4 -> class name(s)
f16: keyword1, keyword2, keyword3, keyword4, keyword5 -> class name(s)
f17: class-name, version, function-name, function-argument-type ->
                                precondition, postcondition, description
f18: class-name, version, function-name, function-argument-type ->
                                function-type, return-type, access-control
f19: function-name -> class-names that contain a function with the same
                      name as the input
f20: class-name, version, attribute-name -> attribute-type, access-control,
                                            description
f21: class-name -> versions
f22: class-name, version, function-name -> function-argument-type-list
-------------------------------------------------------------------------
```

Figure 19: A list of the basic functions

```
EXEC SQL INCLUDE SQLCA;
          :

/* The f1_result table */
EXEC SQL DECLARE f1_result_table TABLE
   (Fname             char(15)   NOT NULL,
    Argu_type_list }  char(30)   NOT NULL);
          :

/* Initialise the database */
          :

/* the implementation of the function f1 */
F1( char *class_name )
{
  EXEC SQL BEGIN DECLARE SECTION;
     STRUCT f1_result_ {  /* corresponding to the 'F1_result_table'  */
        char Fname[16];
        Argument_types[31];
     } f1_result;
     char *cname = class_name;
  EXEC SQL END DECLARE SECTION;

  EXEC SQL DECLARE memfuncsr CURSOR FOR
     SELECT Fname, Argu-type-list
     FROM   MEMFUNC
     WHERE  Cname = :cname
  /* all errors from this point on close down the application */
  EXEC SQL WHENEVER SQLERRER CALL Close_Down;
  EXEC SQL WHENEVER NOT FOUND GOTO Close_memfun_csr;

  EXEC SQL OPEN memfuncsr;

  while (sqlca.sqlcode == 0) /* until no more available record */
  {
    EXEC SQL FETCH memfuncsr INTO :f1_result;
    EXEC SQL INSERT INTO f1_result_table
      VALUES(:f1_result.Fname, :f1_result.Argument_types);
  }
Close_memfun_csr:
 EXEC SQL WHENEVER NOT FOUND CONTINUE;
 EXEC SQL CLOSE memfuncsr;
          :

/* Close down the database */
          :
```

Figure 20: An simplified implementation of the basic function $f1$

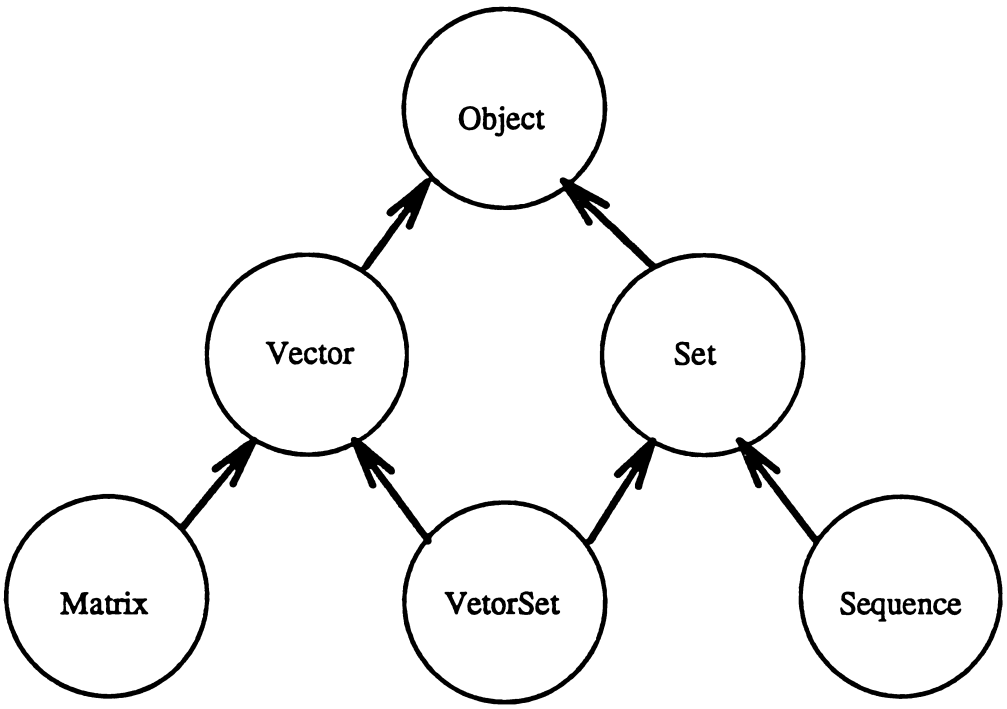


Figure 21: An example hierarchy diagram

that can be dealt with through the menu front-end can be easily expanded just by adding a new basic-function or a new combination of basic functions in the SQL-base.

4.4.2 Algorithm for Finding a Functional Interface of a Given Class

In this subsection, a simple algorithm that the 'query-generator' employed to calculate a functional interface of a given class is introduced. The problem of calculating the interface of a given class which is related to other classes in inheritance relationships, has already been discussed in section 3.2. Figure 21 shows one example of a class hierarchy diagram. A hierarchy diagram can be represented using an acyclic directed graph (DAG), $G = (N, E)$, where N is a set of classes

in a hierarchy graph, and E is a set of inheritance relationships between classes. For example, in figure 21, the class $VetorSet$ is derived from the class $Vector$ and Set. The algorithm for finding all methods which a class, say $VectorSet$ in figure 21, can access is :

1. Find the methods of $VectorSet$ itself and put them into an empty set, say M.
2. Visit the base classes of $VectorSet$ by following the edges from $VectorSet$.
3. Find the methods of the base classes and add them into the set M.
4. Again, move up to the next level, $Object$ in this example, by following the edges from the classes in the current level.
5. Find the methods and add them to the set M.
6. At this point where there are no edges to follow up, we stop the process.

The result of the set M contains all the methods that the class $VectorSet$ can access, i.e.the interface of the class. The following is a formally specified algorithm.

Algorithm for Finding an Interface of a Class

Let q be the class being questioned.
Let R be the relation representing inheritance relationship:
R(superclass,class)
Let A be the relation representing the mapping from class to methods.

A(class,method)

Input

A relation for class hierarchy R, a relation for method definitions A, and a class q being queried.

Output

A set S of methods which the class in question can access(interface).

```
1.  /* create table 'S' for storing the output */
    EXEC SQL CREATE TABLE S
    (member_name  char(20)   NOT NULL);
2.  /* create tables 'temp1' and 'temp2' for temporarily */
    /* storing the superclasses of each level.          */
    EXEC SQL CREATE TABLE temp1
    (class  char(20)   NOT NULL);
    EXEC SQL CREATE TABLE temp2
    (class  char(20)   NOT NULL);
3.  EXEC SQL INSERT INTO S
          SELECT method
          FROM A
          WHERE class = q;
4.  EXEC SQL INSERT INTO temp1
          SELECT superclass
          FROM R
          WHERE class = q;
```

```
5.  WHILE (temp1 is not empty)

6.    EXEC SQL INSERT INTO S
            SELECT A.method
            FROM   A, temp1
            WHERE  A.class = temp1.class;
7.    EXEC SQL INSERT INTO temp2
            SELECT *
            FROM   temp1;
8.    EXEC SQL DELETE FROM temp1
9.    EXEC SQL INSERT INTO temp1
            SELECT R.superclass
            FROM   R, temp2
            WHERE  R.class = temp2.class;
10. END(of while)
```

We assumed that the methods here are all public members because only the public members in base classes can be the methods of its derived class in a client of the derived class's view. The other methods also can be easily calculated using simple conditional statements.

This simple algorithm climbs up one level in a hierarchy graph per loop. If the depth from the top-level class of a hierarchy graph to the class q in question is d, all the methods accessible from the class q can be calculated only in d times of the while loop. The time efficiency of the algorithm is $O(rd)$, where r means

the time efficiency for executing a relational database operation. This operation could be optimised by the *optimizer* in the database system. One advantage of this algorithm is the small number of database access which is the most time-consuming part. For each loop in the algorithm, we need to access the database twice: means $O(d)$ times of accesses for the complete calculation.

4.5 Implementation of the Other Components

Report Generators

Every kind of report-generator is easily implemented by using the the built-in INGRES/REPORT [SKWH76]. INGRES/REPORT is the INGRES subsystem that supports the production of reports in the format of what user wants through either *Report-Writer* or *Report-By-Forms.*

System Maintenance Routine:Concurrency Control

The maintenance involves security and concurrency control support. Since CLIMS is developed for multi-user environments; that is, it is a system that allows any number of transactions[7] to access CLIMS database at the same time, concurrency control mechanism is necessary to ensure that multiple concurrent operations are executed in a manner such that some level of data integrity can be guaranteed. The concurrency control in CLIMS could be simple due to the following reasons:

[7] A transaction is a logical unit of work.

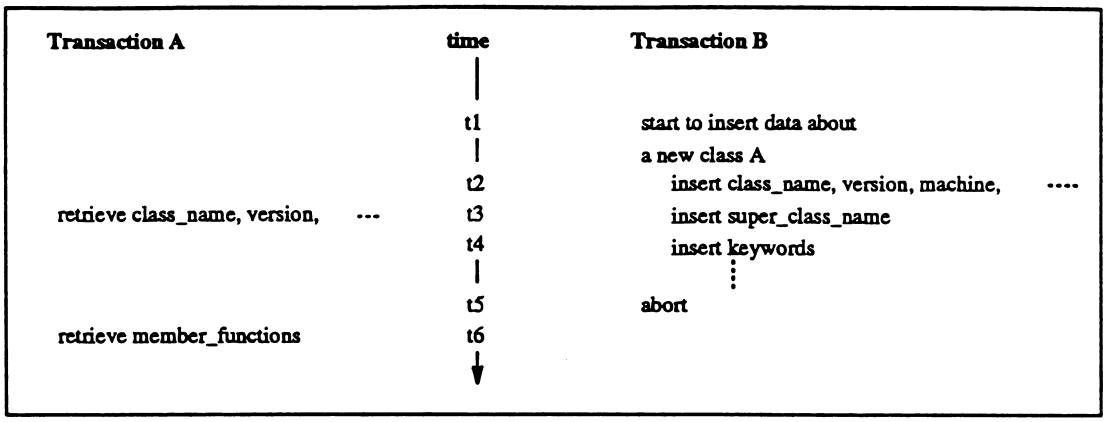


Figure 22: Transaction A becomes dependent on an uncommitted change

- Every update operation on CLIMS is done only through a system administrator.
- Most update operations results from the version-up of a class: i.e. not replacing existing data but inserting new data[8].

However, there still needs a concurrency control. Following is a situation which illustrates a way in which a transaction, though correct in itself, can nevertheless produce the wrong answer because of interference on the part of some other transaction. In figure 22, a client issues the transaction *A* to locate a class for his own use, and find the class A at time t3. The transaction *B* is issued by the 'extractor' to save the reuse information, table by table, about a new class called A. To assure that the class A is reusable, the client tried to retrieve the member functions of the class A at time t6. This retrieval operation fails because the class A does not exist in database any more since the *abort* of transaction *B*.

In CLIMS, the INGRES transaction control statements, *commit* and *rollback*,

[8]Through this way, the current clients of CLIMS are not affected by the update.

are used to insure that simultaneously executing transactions do not interfere with each other. None of the effects on a database of one user's transaction is visible to other users' transactions until the transaction is committed. The principle of concurrency control in CLIMS is that the processing of a single class should be implemented as a single transaction. The problem of the above example can be easily solved by grouping the operations in transaction B into a single transaction such as following:

```
BEGIN TRANSACTION
    insert class_name, version, machine,...
    insert superclass_names
    insert keywords
        .
        :
END TRANSACTION
```

The 'BEGIN TRANSACTION' is implicitly meant by the execution of the first SQL statement. The 'END TRANSACTION' is explicitly implemented by the *commit* command. Queries between *commits* will accumulate as part of the transactions and *locks* [SKWH76] on data accessed by each query will be held until the next commit statement.

4.6 Summary

The implementation of CLIMS has been discussed in detail in this chapter.. Each module, which has been discussed in the implementation plan in the design phase

(see figure 12 and figure 13), is implemented one by one. All the functions rather than the extractor and control system are developed by utilising the INGRES built-in facilities.

Extractor is developed by attaching action-routines to the existing application codes, i.e. specification files for Lex and Yacc.

In the implementation process of the control system, the query-generator is a major part which transforms the user-query from menu selections into the embedded-SQL codes. The SQL-base is introduced in order to facilitate the construction of complex SQLs. The algorithm, which makes it possible for CLIMS to produce the flat-form of a class, is also discussed.

In the implementation process of the system maintenance routine, the concurrency control scheme in CLIMS is discussed briefly.

Chapter 5

Case Study

This chapter is to demonstrate our approach to tackling a complex library system. We have chosen one particular section of a library to highlight the applicability of CLIMS. Nonetheless, this small cross-section is sufficient enough to provide the reader with an insight into the working of CLIMS.

5.1 An Example Library Classes in C++

A set of classes which is chosen for our case study is in figure 23. The classes here can be viewed as a section of a more sophisticated class hierarchy system. The class hierarchy-diagram for the [figure 23] is represented in figure 24. The example illustrates C++'s ability to capture the key concepts of object-orientation. Central to C++ is the *class*, which encapsulates a user-defined type into a single module. The class contains a specification of the data needed to represent an object of the type and a set of operators for manipulating such objects. This

constitutes the *abstraction* of the user-defined type. It also provides different levels of *information hiding* through its use of keywords such as *private, protected* and *public* as shown in *Point* and *Line* classes in the example.[1] The *private* part holds information that can be used only by implementers, while the *protected* part contains information that has a limited right of usage and the *public* part represents an interface with the clients of the type. *Inheritance*(both single and multiple) is also supported as shown in the class *Picture*, where it allows for both behavioural modification as well as type inheritance. The member functions can be made polymorphic by appending the *virtual* keyword to it. *Generic class* is supported by the *template class* as shown in the class *Stack*.

5.2 Retrieval Operations of CLIMS

Running the extractor by taking the [figure 23] as input populates the prepared base tables inside INGRES. The result tables are in appendix B.

Once the library is populated, a user can access the library interface system and does a querying on the system. Following are some typical user-queries and corresponding procedures that generate the answers for the queries. The procedures are represented by using the notation of the procedures in the SQL-base in subsection 4.4.1.

[1]The default is private; i.e. the variables *xc* and *yc* are private members.

```
class Point {
   int xc,yc;   // x-y coordinates
public:
   Point() {xc = yc = 0;};
   int x() const { return xc; };
   int x(int newx) { return xc=newx; };
   int y() const {return yc;};
   int y(int newy) { return yc=newy; };
};
class Shape {
    Point org;       //origin
public:
    Shape(const Point& p) : org(p)  {};
    int orgx() const  { return org.x(); };
    int orgy() const  { return org.y(); };
    virtual void move(int dx, int dy)};
    virtual void draw() const =0;
};
class Line :public Shape {
protected:
   Point p;       // end point
public:
   Line(const Point& a, const Point& b) : Shape(a), p(b) {};
   virtual void move(int dx, int dy) { Shape::move(dx,dy); p.x(p.x()+dx); p.y(p.y()+dy); };
   virtual void draw() const ;
};
class Circle : public Shape {  // Derived from class Shape
   int rad;     // radius of circle
public:
   Circle(const Point& c, int r) : Shape(c)   { rad = r; };
   virtual void draw() const ;
   void Expand(int ExpandBy)   { rad += ExpandBy;};
   void Contract(int ContractBy)  { rad -= ContractBy;};
};
class Arc : public Circle {
   int StartAngle;
   int EndAngle;
public:
   Arc(const Point& c, int r, int InitStartAngle, int InitEndAngle)
            : Circle (c,r) { StartAngle = InitStartAngle; EndAngle = InitEndAngle;};
   virtual void draw() const ;
};
template<class T, int size>  // Generic class for Stack
class Stack {
   T* s[size];
   T** top;
   int sz;
public:
   Stack()   { top=s; sz=0; };
   ~Stack()  { delete s;  };
   void push(T& a)     { sz++; *top++ = &a; };
   T& pop()   { sz--;return **--top; };
   int size() const { return sz; };
};
Stack<Shape,50>  Pstack;   // define a new class Pstack for pictures
class Picture : public Shape, public Pstack  {
   int n;  // number of shapes in this picture
public:
   Picture() : Shape(Point(0,0)), Pstack() { n = 0; };
   Picture( Point& org ) : Shape(org), Pstack() { n = 0; };
   void add(Shape& t)  { Pstack::push(t); n++; };
   virtual void draw() const ;
};
```

Figure 23: Class definitions that illuminate OO concepts in C++

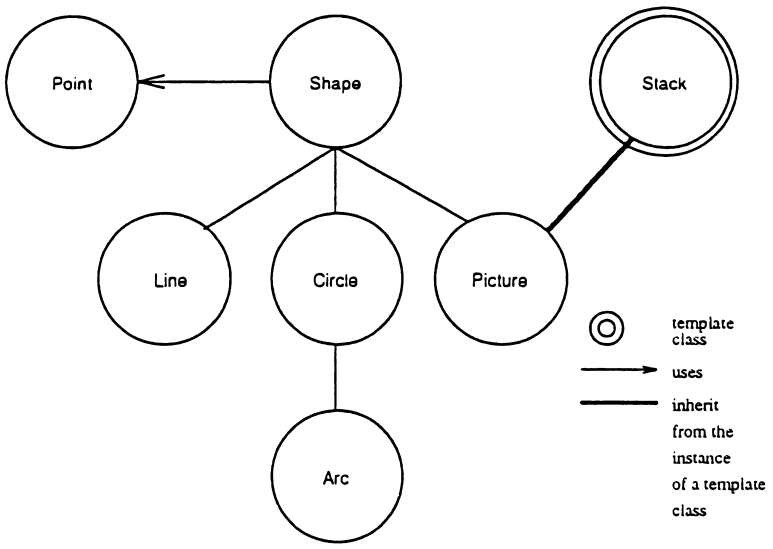


Figure 24: The class hierarchy diagram

1. For an end-user, given certain keywords, search for all the classes that can be utilised.

   EXAMPLE 1: To search for all classes with keywords 'graphics' and 'circle'.

   The embedded-SQL queries are designed by the *query-generator* as:

   call $f13(graphics, circle)$

   display the table $f13\_result\_table$

   which returns the result like following:

classname
Circle

2. For software developers, given a function name, search for all classes that define the function.

   EXAMPLE 2: To search for all the class names that define the public function 'draw'.

Cname	Version	Machine	Compiler	Category	Files	- - -
Shape	1.0	IBM PC	borland C++ 2.0	graphics	gs.cpp	- - -
Arc	1.0	IBM PC	borland c++ 2.0	graphics	gs.cpp	- - -
circle	1.0	IBM PC	borland c++ 2.0	graphics	gs.cpp	- - -
Line	1.0	IBM PC	borland c++ 3.0	graphics	gs.cpp	- - -
Picture	1.0	IBM PC	borland c++ 3.0	graphics	gs.cpp	- - -
Point	1.0	IBM PC	borland c++ 2.0	graphicsa	gs.cpp	- - -
Shape	1.0	IBM PC	borland c++ 2.0	graphics	gs.cpp	- - -
Stack	1.0	IBM PC	borland c++ 2.0	data structure	ds.cpp	- - -

Scname	Cname
Circle	Arc
Shape	Circle
Shape	Line
Shape	Picture
Stack	Picture

Cname	UsedCname
Line	Point
Shape	Point

Figure 25: An example of populated tables: Class, Inhrlt, Use

The embedded-SQL queries are designed by the *query-generator* as:

call $f19(draw)$

display the table $f19\_result\_table$

Thus, the results are:

classname
Shape
Line
Circle
Arc

3. For both the end-user and software developer, given a class name, search for all features belonging to that class.

EXAMPLE 3: To search for detailed description of class "circle".

The embedded-SQL queries are designed by the *query-generator* as:

call $f1(circle)$

call $f2(circle)$

$\vdots$

call $f10(circle)$

display the tables $f1\_result\_tablle, f2\_result\_table, \cdots, f10\_result\_table$

which returns a set of result tables and so the displayer may restructure the format of the output like following:

public member functions	Circle(const Point, int), draw(), Expand(int), Contract(int)
protected attributes	P
class-type	normal class
files	diagram.h, diagram.cpp
machine	apollo2500
compiler	
version	1.0
category	graphics
description	
base class	Shape
sub-class	Arc

4. For all users, given a class name, search for all its superclasses.

   EXAMPLE 4: Select superclasses of the class "arc".

   The procedure is:

   call f8(arc)

   display *f8_result*

   It returns the following result:

superclass
Circle

Chapter 6

Conclusion and Further Work

6.1 Conclusion

In object-oriented (OO) software development, object classes are provided by software vendors as well as developed by application programmers. As the number of classes increases in a multi-user distributed environment, a class library system is urgently needed for the central management of these classes. In this thesis, we have illustrated a relational data dictionary approach to the management of a complicated class library for OO development using C++. Here the data dictionaries are used to describe the reuse information about class definitions and a relational database management system is used for the implementation of a class library.

As mentioned earlier, in order for software reusability to take off, an efficient and reliable library management system is crucial. The proposed approach is

convenient and efficient for the management of class libraries as it allows users to retrieve the classes using non-procedural query languages. There is little overhead required since most existing development environments already provide a database management system, which makes the implementation of the class library management system fairly easy. This compares favourably with the need to develop one's own browsing system. Not only is this effort unnecessary but even if one possesses such a system, the limitations it imposes soon outstrip its short term benefits, whereas the proposed approach has been tested on a fairly complex library system and shown to be economical and efficient in managing the class library for OO development.

6.2 Further Work

Several aspects of CLIMS require further work to achieve the final system. As the system evolves, CLIMS requires an efficient mechanism to support evolution of the system. The prototype system could allow changes without affecting current clients of the class library through the *version-up* technique: i.e. produce a new version of a class and process it by extractor. However, more intelligent tool is required to facilitate the progressive removal of obsolete classes without affecting current clients.

The proposed CLIMS is implemented with a menu-driven user interface. Further work is needed in order to provide users with a graphic display which outlines the hierarchy of class library structure.

Appendix A

Implementation of Basic Functions

```
#include          <stdio.h>

EXEC SQL INCLUDE SQLCA;

/* The Class table */
EXEC SQL DECLARE Class TABLE
(Cname char(20),   /* class name */
         Version char(5),  /* version of class */
         Machine char(20), /* machine used for developing class */
         Compiler char(20), /* compiler used for developing class */
         Category char(20), /* application domain of class */
         Files char(20),    /* files that need to be linked */
         Class_type char(1), /* normal or generic class? */
         Cdescription char(100)); /* brief description of class */
/* The Inhrlt table */
```

```
EXEC SQL DECLARE Inhrlt TABLE
(Scname char(20), /* super class name */
 Cname char(20)); /* class name */
/* The Use table */
EXEC SQL DECLARE Use TABLE
(Cname char(20),
         UsedCname char(20));  /* used class name */
/* The Keywords table */
EXEC SQL DECLARE Keywords TABLE
(Cname char(20),
   Ckwd  char(20)); /* class keyword for future reference */
/* The MemFunc table */
EXEC SQL DECLARE MemFunc TABLE
(Cname char(20),
         Version char(5),
         Fname char(20), /* member function name*/
         Argu_type_list  varchar(50), /* argument list of function */
         Re_type char(15), /* return type of function */
         Access char(7),   /* access control of function */
         F_types char(1), /* normal, virtual, or constant function? */
 PreCondition varchar(50), /* formally specified precondition of ft. */
         PostCondition varchar(50),/* formally specified postcondition of ft. */
         Fdescription varchar(100)); /* brief description of function */

/* The FuncArgu table */
EXEC SQL DECLARE FuncArgu TABLE
(Cname char(20),
         Version char(5),
         Fname char(20),
         Argu_type_list  varchar(50),
         ArguName char(20),  /* name of argument */
```

```
        ArguType char(15)); /* type of argument */
/* The MemAttr table */
EXEC SQL DECLARE MemAttr TABLE
(cname char(20),
        Version char(5),
        Aname char(20),
        Atype char(15),
        Access char(7), /* access control of attribute */
        Adescription varchar(100)); /* brief description of attr. */

/* The f1_result table */
EXEC SQL DECLARE f1_result_table TABLE
   (Fname             char(20),
    Argu_type_list }  char(50));
/* The f2_result table */
EXEC SQL DECLARE f2_result_table TABLE
   (Aname  char(20),
    Atype  char(15));
/* The f3_result table */
EXEC SQL DECLARE f3_result_table TABLE
   (Fname             char(20),
    Argu_type_list }  char(50));
/* The f4_result table */
EXEC SQL DECLARE f4_result_table TABLE
   (Aname  char(20),
    Atype  char(15));
/* The f5_result table */
EXEC SQL DECLARE f5_result_table TABLE
   (Fname             char(20),
    Argu_type_list }  char(50));
/* The f6_result table */
```

```
EXEC SQL DECLARE f6_result_table TABLE
   (Aname  char(20),
    Atype  char(15));
/* The f7_result table */
EXEC SQL DECLARE f7_result_table TABLE
(Machine char(20),
         Compiler char(20),
         Category char(20),
         Files char(20),
         Class_type char(1),
         Cdescription);
/* The f8_result table */
EXEC SQL DECLARE f8_result_table TABLE
(SuperCname char(20)); /* class name */
/* The f9_result table */
EXEC SQL DECLARE f9_result_table TABLE
(SubCname char(20)); /* class name */
/* The f10_result table */
EXEC SQL DECLARE f10_result_table TABLE
(UsedCname char(20)); /* class name */
/* The f11_result table */
EXEC SQL DECLARE f11_result_table TABLE
(Cname char(20)); /* class name */
/* The f12_result table */
EXEC SQL DECLARE f12_result_table TABLE
(Cname char(20)); /* class name */
/* The f13_result table */
EXEC SQL DECLARE f13_result_table TABLE
(Cname char(20)); /* class name */
/* The f14_result table */
EXEC SQL DECLARE f14_result_table TABLE
```

```
(Cname char(20)); /* class name */
/* The f15_result table */
EXEC SQL DECLARE f15_result_table TABLE
(Cname char(20)); /* class name */
/* The f16_result table */
EXEC SQL DECLARE f16_result_table TABLE
(Cname char(20)); /* class name */
/* The f17_result table */
EXEC SQL DECLARE f17_result_table TABLE
(PreCondition varchar(50),
         PostCondition varchar(50),
         Fdescription varchar(100)); /*  */
/* The f18_result table */
EXEC SQL DECLARE f18_result_table TABLE
(Re_type char(15), /* class name which the function belong to */
         Access char(7),
         F_types char(1)); /*  */
/* The f19_result table */
EXEC SQL DECLARE f19_result_table TABLE
(Cname char(20)); /* class name */
/* The f20_result table */
EXEC SQL DECLARE f20_result_table TABLE
(Atype char(15), /* class name which the attribute belong to */
         Access char(7),
         Adescription varchar(100)); /* function name if the attribute is from the functi
/* The f21_result table */
EXEC SQL DECLARE f21_result_table TABLE
(Version char(5)); /*  */
/* The f22_result table */
EXEC SQL DECLARE f22_result_table TABLE
(Argu_type_list  varchar(50)); /*  */
```

```
/* Procedure: Init_Db
** Purpose: Initialize the database.
**          Connect to the database, and abort if an error.
*/
void Init_Db()
{
  EXEC SQL WHENEVER SQLERROR STOP;
  EXEC SQL CONNECT CLIMS_DB;  /* connect to the CLIMS */
}
          .
          .
          .

/* Procedure: F1
** Purpose: Implementate the basic function f1 in SQL-base
*/
F1( char *class_name, char *version )
{
  EXEC SQL BEGIN DECLARE SECTION;
     STRUCT f1_result_ {  /* corresponding to the 'f1_result_table'  */
        char Fname[21];
        char Argument_types[51];
     } f1_result;
     char *cname = class_name;
     char *ver = version;
  EXEC SQL END DECLARE SECTION;

  EXEC SQL DECLARE memfuncsr CURSOR FOR
     SELECT Fname, Argu-type-list
```

```
    FROM   MemFunc
    WHERE  Cname = :cname
      AND  Version = :ver
      AND  Access = 'public';
  /* all errors from this point on close down the application */
  EXEC SQL WHENEVER SQLERROR CALL Close_Down;
  EXEC SQL WHENEVER NOT FOUND GOTO Close_memfun_csr;

  EXEC SQL OPEN memfuncsr;

  while (sqlca.sqlcode == 0) /* until no more available record */
  {
    EXEC SQL FETCH memfuncsr INTO :f1_result;
    EXEC SQL INSERT INTO f1_result_table
      VALUES(:f1_result.Fname, :f1_result.Argument_types);
  }

Close_memfun_csr:
 EXEC SQL WHENEVER NOT FOUND CONTINUE;
 EXEC SQL CLOSE memfuncsr;
}
/* Procedure: F2
** Purpose: Implementate the basic function f2 in SQL-base
*/
F2( char *class_name, char *version)
{
  EXEC SQL BEGIN DECLARE SECTION;
     STRUCT f2_result_ {  /* corresponding to the 'f2_result_table'  */
        char Aname[21];
        char Atype[16];
     } f2_result;
```

```
    char *cname = class_name;
    char *ver = version;
  EXEC SQL END DECLARE SECTION;

  EXEC SQL DECLARE memattrcsr CURSOR FOR
    SELECT Aname, Atype
    FROM   MemAttr
    WHERE  Cname = :cname
      AND  Version = :ver
      AND  Access = 'public';
  /* all errors from this point on close down the application */
  EXEC SQL WHENEVER SQLERRER CALL Close_Down;
  EXEC SQL WHENEVER NOT FOUND GOTO Close_memfun_csr;

  EXEC SQL OPEN memattrcsr;

  while (sqlca.sqlcode == 0) /* until no more available record */
  {
    EXEC SQL FETCH memattrcsr INTO :f2_result;
    EXEC SQL INSERT INTO f2_result_table
      VALUES(:f2_result.Aname, :f2_result.Atypes);
  }
}
/* Procedure: F3
** Purpose: Implementate the basic function f3 in SQL-base
*/

F3( char *class_name, char *version )
{
  EXEC SQL BEGIN DECLARE SECTION;
    STRUCT f3_result_ {  /* corresponding to the 'f3_result_table'  */
```

```
        char Fname[21];
        char Argument_types[51];
    } f3_result;
    char *cname = class_name;
    char *ver = version;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE memfuncsr CURSOR FOR
    SELECT Fname, Argu-type-list
    FROM   MemFunc
    WHERE  Cname = :cname
      AND  Version = :ver
      AND  Access = 'protected';
/* all errors from this point on close down the application */
EXEC SQL WHENEVER SQLERROR CALL Close_Down;
EXEC SQL WHENEVER NOT FOUND GOTO Close_memfun_csr;

EXEC SQL OPEN memfuncsr;

while (sqlca.sqlcode == 0) /* until no more available record */
{
  EXEC SQL FETCH memfuncsr INTO :f3_result;
  EXEC SQL INSERT INTO f3_result_table
    VALUES(:f3_result.Fname, :f3_result.Argument_types);
}
}
        .
        .
        .
```

```
/* Procedure: Close_Down
** Purpose: Error handler called any time after Init_Db has been
**          successfully completed. In all case, print the cause of
**          the error and abort the transaction, backing out
**          changes.
*/
Close_Down()
{
    EXEC SQL BEGIN DECLARE SECTION;
            char   errbuf[101];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL WHENEVER SQLERROR CONTINUE; /* Turn off error handling */

    EXEC SQL INQUIRE_INGRES (:errbuf = ERRORTEXT)
    printf("Closing Down because of database error:\n")
    printf("%s\n", errbuf);

    EXEC SQL ROLLBACK;
    EXEC SQL DISCONNECT
    EXIT( -1 );
}
/* Procedure: End_Db
** Purpose: Commit the multi-statement transaction and access to
**          the database.
*/
void End_Db()
{
  EXEC SQL COMMIT;
  EXEC SQL DISCONNECT;
}
```

Bibliography

[AM88] W.W. Agresti and F.E. McGarry. The minnowbrook workshop on software reuse:a summary report. pages 33–40, 1988.

[AS87] S.P. Arnold and S.L.Stepoway. The reuse system:cataloging and retrieval of reusable software. *Proceedings of COMPCON*, pages 376–379, Summer 1987.

[BBJR87] V. Basili, J. Barley, B. Joo, and H. Romback. Software reuse: A framework. *Minnowbrook Workshop on Software Reuse*, 1987.

[Boe86] B. Boehm. A spiral model of software development and enhancement. *Software Engineering Notes*, 11(4), 1986.

[Boo91] G. Booch. *Object-Oriented Design with Application.* The Benjamin/Cummings Pub.Co. 1991., 1991.

[BRB+87] B.A. Burton, R.W.Aragon, S.A. Bailey, K.D. Koehler, and L.A. Mayers. The reusable software library. *IEEE Software*, pages 25–33, July 1987.

[BSJ90] B.Henderson-Sellers and J.M.Edwards. The object-oriented systems life cycle. *Comm. ACM*, 33(9):142–159, 1990.

[Cod70] E. Codd. A relational model of data for large shared data bank. *CACM*, 13(6), june 1970.

[Cox91] B. J. Cox. *Object-Oriented Programming: an Evolutionary Approach.* Addison-Wesley Publishing Company, 2nd edition, 1991.

[CW85] L. Cardelli and P. Wegner. Understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[Dat83] C. J. Date. *An Introduction to Database Systems*, volume 2. Addison-Wesley Publishing Company, 1983.

[Dat90] C. J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley Publishing Company, 5th edition, 1990.

[Dav83] W. S. Davis. *System Analysis and Design.* Addison-Wesley Publishing Company, Inc., 1st edition, 1983.

[Dem79] T. Demarco. *Structured Analysis and System Specification.* Yourdon Press. Prentice-Hall Company, 1st edition, 1979.

[DF76] D.Tsichritzis and F.Lochovsky. Hierarchical data base management : A survey. *ACM Computing Surveys*, 8(2), March 1976.

[DMS89] D.Taenzer, M.Ganti, and S.Podar. Object-oriented software reuse:the yoyo problem. *JOOP*, pages 30–35, September/October 1989.

[emb89a] *INGRES/Embedded SQL COMPANION GUIDE FOR C*, release6.2 unix edition, August 1989.

[emb89b] *INGRES/Embedded SQL User's Guide and Reference Manual*, release 6.2 unix edition, August 1989.

[FPDMS91] W.B. Frakes, R. Prieto-Diaz, K. Mastsumura, and W. Schaefer. Software reuse: Is it delivering? *1991 IEEE 13th International Conference on Software Eng.*, pages 52–59, 1991.

[Gor87] K. E. Gorlen. An object-oriented class library for c++ programs. *Software-Practice and Experience*, 17(12):899–922, 1987.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Mass., 1983.

[GTNP90] S. Gibbs, D. Tsichritzis, O. Nierstrasz, and X. Pintado. Class management for software communities. *Comm. ACM*, 33(9), Sept. 1990.

[HHKM91] H.M.Al-Haddad, K.M.George, and M.H.Samadzadeh. Approaches to reusability in c++ and eiffel. *JOOP*, pages 34–45, September 1991.

[Jon84] T.C. Jones. Reusability in programming:a survey of the state of the art. *IEEE Trans. Software Eng.*, pages 488–494, Sept 1984.

[KL92] G. Kiczales and J. Lamping. Issues in the design and specification of class libraries. *OOPSLA*, pages 435–451, 1992.

[KM92] T. Korson and J.D. McGregor. Technical criteria for the specification and evaluation of object-oriented libraries. *Software Engineering Journal*, pages 85–94, March 1992.

[KS91] H.F. Korth and A. Silberschatz. *Database System Concepts.* McGraw-Hill,Inc, 2nd edition, 1991.

[LVC89] M.A. Linton, J.M. Vlissides, and P.R. Calder. Composing user interfaces with interviews. *IEEE Computer*, pages 8–22, Feb. 1989.

[Ma91] J. Ma. *An Object-Oriented Approach to Model Management.* PhD thesis, Asian Institute of Technology, Bangkok, Thailand, 1991.

[MB90] T. Mason and D. Brown. *Lex & Yacc.* O'Reilly & Associates, Inc, 1990.

[MBK91] Y.S. Maarek, D.M. Berry, and G.E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE TRANSACTION OF SOFTWARE ENG.*, 17(8):800–813, Aug 1991.

[Mey87] B. Meyer. Reusability:the case for object-oriented design. *IEEE Software*, pages 50–64, March 1987.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[Mey90] B. Meyer. Lessons from the design of the eiffel libraries. *Comm. ACM*, 33(9):69–88, Sept. 1990.

[Mey92] B. Meyer. *Eiffel: The Language.* Prentice-Hall Inc., 1992.

[NMN93] Kek Wee Ng, Jian Ma, and Gi-Moon Nam. Class library management system for object-oriented programming. *Proceedings of SAC'93,ACM*, pages 445–451, 1993.

[P.P76] P.P.Chen. The entity-relationship approach to logical data base design. *ACM Transactions on Data Base Systems*, 1(1), January 1976.

[PP92] A. Podgurski and L. Pierce. Behavior sampling: A technique for automated retrieval of reusable components. *14th proceedings of Conference on Software Engineering*, 1992.

[RDF87] R.Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, pages 6–16, January 1987.

[Ros90] J.A. Roskind. Cpp4.y & cpp4.l. as Public Domain Source Code, 1990.

[SEHVG91] S.K.Abd-El-Hafiz, V.R.Basili, and G.Caldiera. Towards automated support for extraction of reusable components. *1991 IEEE Conference on Software Maintenance*, pages 212–219, 1991.

[SKWH76] M. Stonebraker, P. Kreps, E. Wong, and G. Held. The design and implementation of ingres. *ACM Transactions on Database System*, 1(3), september 1976.

[Str91] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Mass., 2nd edition, 1991.

[TYF86] T. J. Teorey, D. Q. Yang, and J. P. Fry. A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model. *ACM Computing Surveys*, 18(2):197–222, June 1986.

[WB87] W.B.Frakes and B.A.Nejmeh. An information system for software reuse. *Proceedings of the 10th Minnowbrook Workshop on Software Reuse*, 1987.

[WBJ90] R. J. Wirfs-Brock and R. E. Johnson. Surveying Current Research In Object-Oriented Design. *Comm. ACM*, 33(9):105–124, Sept 1990.

[Weg90] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1):7–87, Aug 1990.

[WI88] M. Wood and Sommervile I. An information retrieval system for software components. *Software Engineering Journal*, Sep 1988.

[WJ90] W.R.LaLonde and J.R.Pugh. *Inside Smalltalk.* Prentice Hall, Englewood Cliffs, N.J., 1990.

[Wu90] C. T. Wu. A better browser for object-oriented programming. *Journal of Object-Oriented Programming*, pages 22–28, Nov/Dec 1990.

[Yao85] S. Yao. *Principles of Database Design Vol 1: Logical Organizations.* Prentice-Hall, Englewoord Cliffs, NJ, 1985.

[Zin90] Inc. Zinc. *Zinc Reference Manual*, 1990.