A Repository of Software Components

by

Wanjie Wang

A thesis

Submitted to the Faculty of Graduate Studies

in Partial Fulfillment of the Requirements

for the degree of

MASTER OF SCIENCE

Department of Computer Science

University of Manitoba

Winnipeg, Manitoba, Canada

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

0-612-41647-X

Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

A Repository of Software Components

BY

Wanjie Wang

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree

of

MASTER OF SCIENCE

Abstract

Component-based software development (CBSD) is an enhancement of the object-oriented (OO) approach to software development. The major distinction between a component in CBSD and an object in OO is that the former can be defined in several levels of abstraction whereas objects are mostly defined at the design and code levels. Among the many challenges posed by CBSD, the following two seem to be more important: (i) develop a repository to store, retrieve and manipulate components: and (ii)use the components in a plug-and-play mode to build applications. The work presented in this thesis is a contribution to the first challenge which is to build a repository for supporting the use of software components in the development of generic software architecture. A tool for building such a repository has been implemented in Java, with features to classify, store and retrieve components. By providing precise definitions for classification of components, relationships between components, and the characteristics of components, we assert that such a repository can be used in a variety of application domains. A hybrid approach for retrieval which supports multi-level keywords search and Boolean query search has been developed. The tool has been used to build several prototype repositories; the results are promising.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Kasilingam Periyasamy, for both his technical and moral support, as well as his enthusiasm and encouragement. His suggestions and comments were always so much to the point that I often wondered how I could have missed them.

I would like to thank the members of my thesis committee, Professor Witold Kinsner and Peter Graham, for their constructive comments and suggestions.

I would also like to thank a number of people inside and outside the computer science department who helped me over the years. Special thanks to Klaus Krauter, a member of our research group, for his helpful discussion on the project. Gilbert Detillieux and Tom Dubinski, for their technical support keeping the machine running. Helen Cameron for the "toughest" course I took from her. David Scuse for the help of using Cargill Lab. I particularly want to thank Xiaowei Liu and Wei Liu for the time we spent at the U of M.

I extend my heartfelt thanks to my wife, Yangjie Gong, for her love, support, and faith in me.

Finally, I would like to thank my parents and other family members for their emotional support and encouragement all through.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software reuse is expected to make a breakthrough in software productivity, mainly because a reuse process will save cost and time. Developing a software product from scratch is not only a time consuming and an expensive process, but is also a tedious process. It has been shown [Voa98] that even the best programmers can only deliver 10 lines of well-documented and well-tested code per day. On the other hand, reusing 100,000 lines of code in a project which requires 300,000 lines of code can save severe programmer days.

Reuse can be applied at any stage in the software development life cycle. Code reuse has been in practice in industry for several decades. At the code level, one can see that the code for a software product consists of well-defined algorithmic blocks such as *while* loops and conditional statements. By properly composing these algorithmic blocks in an application-independent fashion, a programmer can develop a generic reusable software product. Code libraries such as packages of mathematical functions are typical examples of code reuse. Although code reuse has been in practice for several years, the benefits achieved through code reuse have been found to be far beyond the expectations.

There are two inherent problems with code reuse: *identification* of a piece of code that can be reused, and *justification* that the code can indeed be reused in the new environment. Identification of a code for reuse requires extensive searching and complex pattern matching algorithms. Even with sophisticated algorithms, a code that behaves in an expected manner may not match the parameter of a reuse query unless there exists a mechanism for proper unification of names in the code and the names in the query. Justification of code reuse is even more difficult; it requires a formal semantics of the programming language in which the reusable product has been implemented. Because of these difficulties in code reuse, software developers have recently been looking for abstract definitions of software products that can be easily identified as well as justified. An obligation for this work is that the abstract definition and the code which implements the definition, should be bundled together so that one can search for and match the abstract definition, and then reuse the associated code without any problems. The notion of *encapsulation* in object-orientation exactly satisfies this requirement, and hence the object-oriented approach to software development is claimed to promote software reuse.

An object-oriented software product consists of a collection of discrete objects which encapsulate data and behaviors. The distinguishing characteristics of object-orientation such as *inheritance, encapsulation* and *polymorphism* help programmers to achieve code reuse at varying levels of granularity. For example, inheritance has been used as a mechanism for code reuse in the AI community. In the object-oriented paradigm, inheritance allows the reuse of both methods and data together. Universal polymorphism, on the other hand, enables a designer to identify and justify reuse at the class level. Encapsulation is the key for matching interfaces of objects hiding their implementation details and hence promoting reuse at the class level.

It has been argued that the benefits of reuse provided by object-oriented program-

ming are still inadequate because object-oriented programming too often concentrates on individual objects, instead of whole collections of objects [Pfi96]. Recently, software developers have shifted their focus from the traditional notion of an "object" to a "component". A *component* is a primitive building block of a software system, serving as a significant functional element of the system. A component can be anything, ranging from a piece of code such as a low-level procedure to a high-level module such as a Management Information Subsystem. In general, components can be implemented in any paradigm, in any programming language, or even with dedicated hardware. For the purpose of this thesis, a software component is defined to be an entity which has a significant role in the development of software. Documents, design diagrams, code, algorithms, and test cases are some examples of software components. The big promise of component-based software development is to reuse previously developed, or off-the-shelf components as much as possible to build new applications more rapidly, economically and reliably.

In this thesis, we define a *software components repository* as a library that contains software components. As a component resource, the repository can allow software engineers to (i) browse the existing components as a source of examples, (ii) extract the components and (iii) compose components to build larger systems.

The first challenge in component-based software development (CBSD) is therefore the ability to locate a component in the repository. It should be significantly easier and faster to find a component than to write it from scratch. A component can be found either by describing it with a search or query and inspecting the results of the query, or by browsing through the repository possibly based on some criteria until the desired component is found. The second challenge is the plug-and-play mode of operation for components. Once we find the desired components, we must know whether these components can fit into the application under construction. The

components may be used directly, or may have to be modified before being used. The work presented in this thesis is a contribution to the first challenge; i.e., to build a repository for supporting the use of software components in the development of generic software architectures [Bas98].

Previous research on classification, storage, and retrieval of software components mainly falls into four broad categories: *text-based information retrieval, the AI-based semantic network approach, the faceted scheme approach,* and the *formal specification-based approach.*

In text-based information retrieval, keywords are automatically extracted from software documents. An index is then created by associating each component in the library with a set of keywords. This approach uses proven information retrieval and indexing technology. The advantage of this approach lies in its simplicity and possibilities for automation. A limitation of this approach, however, is that the keywords usually do not carry sufficient semantic information. Maarek and others [Maa91] proposed a method using the concept of lexical affinities among pairs of words and their statistical distribution to extract some semantic information from a document automatically. The low cost of building the associated library coupled with adequate performance has made this approach popular in commercial text retrieval systems and WWW engines such as Yahoo and Alta Vista [Hen97].

The use of AI-based semantic networks [Ost92] provides a structural representation of knowledge with some inference capability. It is suitable for representing the concepts contained in software components, but this approach is extremely labor-intensive for creating the required knowledge structure for a system of any significant size. Another disadvantage is that the application domains to which this approach can be effectively applied are too restrictive.

The faceted scheme approach [Pri91, Cha97] relies on a predefined set of keywords,

extracted by experts, from program description and documentation. These keywords are arranged by facets into a classification scheme and used as standard descriptions for software components. In the faceted scheme, similarity between required software specification and available components is evaluated using conceptual distance, which estimates expected effort to adapt a software component to satisfy a given design specification. Using this approach, it is easy to combine keywords to represent components. but the users are required to know the structure of the library and keywords, and to have an understanding of the facets.

Formal specifications-based approaches [Mil94, Ale95, Zar97] use formal methods to describe the behavior of software components. Formal specifications can capture the semantics of software components more precisely and, hence provide more detailed information about components. However. such approaches have similar disadvantages to those of the faceted approach; e.g., there is a steep learning curve associated with the use of formal specifications.

Recently, some significant contributions have been reported in the use of software components [Pod93, Bat97, Hen97]. Mili and others [Mil94, Mil97] discuss an approach based on formal specifications and the refinement ordering between specifications. Zaremski and Wing [Zar95] discuss signature-match as a method for storing and retrieving software components. They introduce different kinds of match predicates: exact match and various relaxed matches for both function and module components. In another paper, Zaremski and Wing [Zar97] have shown an extension of their previous work by investigating specification matching in detail. They use formal specifications, which contain pre- and postconditions to describe the behavior of components, and define a general matching criterion that is applied to a wide selection of specifications. Chang and others [Cha97] present an approach to reuse-based software development using formal methods. In this approach, each component is annotated

with a set of predicates to formally describe the component. The components are classified by the formal predicates using the faceted scheme. After a component is retrieved. it is integrated with the design system, and then the integrated system is transformed into a Predicate/Transition net (PrTnet), which describes the design specifications of the application, to perform consistency checking.

Two other known component-based software development approaches are Microsoft's Component Object Model (COM) [COM94] and the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [COR95].

The Component Object Model (COM) is a software architecture that allows applications to be built from binary software components. It was first introduced by Microsoft in 1993. The core of COM is a specification for how components and their clients interact with each other. In this context, COM defines several fundamental concepts.

1. A binary standard for function calls between components that allows components written in different languages to call each other.

2. A provision for strongly-typed groupings of functions into interfaces. In COM, applications interact with each other and with the system through collections of function headers called interfaces. A COM Interface is a strongly-typed contract between software components to provide a set of semantically related operations (methods).

3. *IUnknown* is a special interface defined by COM to implement some essential functionality. All COM components are required to implement the IUnknown interface. IUnknown has three methods: *QueryInterface*, *AddRef*, and *Release*.

   - *QueryInterface* provides a way for components to dynamically discover the

interfaces supported by other components.

- *AddRef* and *Release* provide facilities to encapsulate a component's lifetime through reference counting.

4. A mechanism to uniquely identify components and their interfaces using GUIDs (globally unique identifiers).

DCOM is the Distributed extension to COM that adds an object remote procedure call layer on top of the Open Software Foundation's DCE-RPC specification to support remote objects [Chu98]. DCOM thus provides client-server type communications. To request a service, a client invokes a method implemented by a remote object: the remote object acts as the server in the client-server model. An Interface Definition Language (IDL) is used to describe the interface of an object. To invoke a remote function, the client makes a call to the *client stub* (called *a proxy*). The stub packs the call parameters into a request message, and invokes a wire protocol to ship the message to the server. At the server-side, the wire protocol delivers the message to the *server stub* (called *a stub*), which then unpacks the request message and makes a call to the actual method on the server object.

The Common Object Request Broker Architecture (CORBA) is a distributed object-middleware system developed by the Object Management Group (OMG), a consortium of over 800 companies. The core of CORBA is the *Object Request Broker* (ORB) that acts as the object bus over which objects transparently interact with other objects located locally or remotely. The ORB intercepts each call from a client object, finds the server object that implements the request, passes the parameters to the corresponding object, invokes its method, and returns the results, if any, back to the client.

A CORBA object is represented by an interface with a group of methods. This

interface is specified in an Interface Definition Language (IDL), which is independent of any programming environment. The IDL compiler generates *stub* code that the client can link to, and *marshals* the programming data types into a wire format for transmission as a request to an object implementation. The implementation of the object has linked to it similar marshaling code, called a *skeleton*, that *unmarshals* the request into programming language data types [Vog98].

Both COM/DCOM and CORBA are object middleware solutions which act as a layer of software that insulates application software from system software and other technical or proprietary aspects of underlying run-time environments. Object models and component models are, however, different. Object models do not contain all the necessary concepts for assembling and describing components.

Currently, both in academia and in industry, component-based software development is progressing rapidly. Research on software components is continuously evolving and branching out in various directions. However, many components repositories still use the file system for their organization (i.e simple directories and files) and file system and editor commands for navigation and retrieval. For example, in a UNIX system, one might use *ls*, *find* or *grep* to locate desired components. The task of finding something in these repositories must rely on the names of components. Unfortunately, it is very hard for a developer to understand the functionality of a reusable component based just on its name. The successful use of a component repository relies on the availability of good tools to organize, navigate through, and retrieve components from repositories. These tools can also help developers to learn, and to familiarize themself with the sophistication of the components.

The goal of this thesis is to develop a tool, named *CSRTool* (the Components Storage and Retrieval Tool), which forms the front-end of a software components repository. This tool can be used to classify, store, and retrieve components. In this

thesis. components are mainly classify in terms of their types and roles. Components will be stored in a hierarchy and multi-level keywords and Boolean query retrieval will initially be used for retrieving components. Using CSRTool, components should be able to be added and/or modified at any time during its use. A type manager, part of the CSRTool. will allow a user to define new types, add new types to the repository as well as modify and/or remove existing types. After a new type is created, a relationship editor will permit the user to add new relationships between the new type and other existing types. Users will also be allowed to modify and remove existing relationships among types. The CSRTool will also support a user-friendly GUI with separate windows for storing and retrieving components. CSRTool will be implemented using the object-oriented approach; UML analysis and design diagrams for the tool architecture will be included in the thesis. The Java language will be used for the implementation so that CSRTool can be used as a stand-alone application. or as an applet on a browser. CSRTool will represent an incremental improvement of component repositories. thereby supporting the development of generic software architectures.

The rest of the thesis is organized as follows: Chapter 2 introduces the terminology relevant to component-based software development that is used in this thesis. In Chapter 3 the architecture of CSRTool is presented and the mechanism used to store and retrieve components using CSRTool is discussed. The mechanisms used to introduce new types for components and to support incremental refinement of the repository are also discussed in Chapter 3. Finally, Chapter 4 summarizes the thesis and discusses continuing work in this direction.

Chapter 2

Software Components

A *software component* is an entity which has a significant role in the development of software. Typical software components are documents describing requirements and designs, design diagrams, code modules, and test cases. From the definition of software components given above, one can infer that a software component may be *atomic* or *composite*. A component may contain other components as its sub-components, which are themselves atomic or composite. For example, a block of code may contain several routines and data structures, each one of them being considered a software component. Therefore, the architecture of a complete system can be described as a hierarchical organization of components. Software components become primitive building blocks of a software architecture, and thus provide the basic infrastructure for a software architecture.

The *software architecture* of a program or a computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them [Bas98]. This indicates that a software architecture not only describes a set of software components, but also describes the interactions among these components. In other words, a software

architecture captures both static and dynamic behavior of the components in the architecture. Therefore, the design of a software system can be thought of as an iterative process with the software architecture describing the initial set of components and their interactions [Bat92, Sha96, Wol97, Kaz97].

2.1 Classification of Components

The successful development of a repository lies in the proper classification of the components in the repository, which in turn requires characterization of the components. The performance of component retrieval algorithms depends heavily on the structural classification of the components stored in the repository [Sha97]. Depending on application domains and hardware platforms, the classification of components will vary. For convenience and simplicity, in this thesis, we provide the following classification of components. Later in the thesis, we will show how to extend this classification to include a wider range of components.

- **File**: A file is a container which may contain other components such as documentation, code, design diagrams or test case.
- **Documentation**: A documentation is a textual description of another component. For example, it may describe a design diagram or implementation details of a model.
- **Class**: The notion of a class in component-based programming is the same as in object-oriented programming. A class represents a group of objects with similar properties (attributes), common behaviors (operations), and common relationships to other objects [Rum91].
- **Routine**: A method or procedure with its signature (interface) and code.

- **Data Structure**: Represents data structures such as arrays, trees, and lists. A data structure may or may not contain the associated code for its manipulation. For example, a linked list may contain the code for inserting, deleting, and retrieving elements from the list while a structure may simply contain only its signature.

2.2 Relationships between Components

A component does not exist in isolation. It is normally related to other components. For instance, a class definition may consist of variables (attributes), data structures and routines. There may exist static relationships between one or more class definitions. It is thus important to identify the relationships of a component with other components and store these relationships along with the definition of the component itself.

Table 2.1 describes a number of relationships between different types of components that were initially identified and listed in the previous section. These types are well-known and have been used for several decades in software development, and so are the relationships enumerated in Table 2.1. Completeness or exhaustivity of this table is beyond the scope of the current research. The CSRTool (to be discussed in detail in subsequence chapters) does provide the facility to add new types and/or new relationships between types. It is therefore possible to extend the set of relationships in Table 2.1 later, depending on the application domain chosen.

Type	File	Documentation	Class	Routine	Data Structure
File	includes contains	contains	contains	contains	contains
Documentation	enclosed-in span-over	references includes	describes	describes	describes
Class	enclosed-in span-over	described-by	aggregates generalizes specializes associates	contains accesses accessed-by	contains
Routine	enclosed-in span-over	described-by	enclosed-in accesses accessed-by	calls called-by enclosed-in	contains accesses accessed-by
Data Structure	enclosed-in span-over	described-by	enclosed-in accesses accessed-by	enclosed-in accesses accessed-by	enclosed-in accesses accessed-by contains

Table 2.1: Relationships between different types of components

The semantics of the relationships are as follows:

X *includes* Y: X has a pointer or a link to Y. The access mechanism (e.g. program pointer or HTML pointer) are part of implementation details. This is somewhat similar but not identical to structural association between classes in object-orientation.

X *contains* Y: X contains an entire copy of Y. This is similar to the *aggregation* relationship between classes in object-oriented but not identical. The *contains* relationship emphasizes more on physical containment, while *includes* does not emphasize physical containment.

X *enclosed-in* Y: Inverse relation of X *contains* Y.

X *span-over* Y: There are several instances of Y (such as Y1, Y2, ..., Yn), each of which *contains* a portion of X.

X *references* Y: X has a pointer to Y. There is a difference between *references* and *includes*: When X *references* Y, the information in Y is not mandatory for the use of X. Information in Y typically enhances the information in X. On the other hand, when X *includes* Y, information in Y is a part of or a component of information in X. Hence, every usage of X also requires a copy of Y.

X *describes* Y: X typically explains about Y (what it is, how to use it and so on).

X *described-by* Y: Inverse relation of X *describes* Y.

X *aggregates* Y: The meaning of this relationship is the same as the *aggregation* relationship between classes in object-orientation. An instance of X is a part of each instance of Y. Aggregation is a special form of association.

X *generalizes* Y: The meaning of this relationship is the same as the generalization relationship between classes in object-orientation.

X *specializes* Y: The meaning of this relationship is the same as the specialization relationship between classes in object-orientation. That is, X is a refinement of Y.

X *associates* Y: The meaning of this relationship is the same as association relationship between classes in object-orientation. X and Y are independent components. An instance of X has a static link to an instance of Y.

X *accesses* Y: An instance of X uses some information in an instance of Y during its existence. There exists a dynamic link between the instance of X and an instance of Y. Existence of the instance of Y is not mandatory for the existence of the instance of X.

X *accessed-by* Y: Inverse relation of X *accesses* Y.

X *calls* Y: This relationship is specific for routines only. The meaning is self-explanatory. We would like to stress that X *calls* Y is different from X *accesses* Y; when X *calls* Y, X uses all information in Y whereas when X *accesses* Y, X may use only a part of Y.

X *called-by* Y: Inverse relation of X *calls* Y.

Using the semantics of these relationships, one can categorize the strength of a relationship ranging from "strong" to "weak". For example, *includes* is stronger than *references* while *includes* is weaker than *contains*. Thus, there exists a hierarchy of strength among these relationships. Such a hierarchy is useful particularly when a component is retrieved using a certain threshold specified in the query. The current version of the CSRTool does not implement this feature; this has been left for future work because the implementation of this feature requires semantic analysis of the components.

Chapter 3

The Components Storing and Retrieving Tool (CSRTool)

In this chapter, the architecture of CSRTool is first presented in Section 3.1. Then the mechanism used to store and retrieve components using CSRTool is discussed in sections 3.2 and 3.3 respectively. In Section 3.4, issues related to defining a new component type and the support of incremental refinement of a repository are presented. The Chapter concludes with a brief discussion of the support provided by the Java language used in implementing CSRTool.

3.1 Architecture of the CSRTool

The general architecture of CSRTool is shown in Figure 3.1. There are two types of users of the tool: *maintainer* and *user*. The users can query and retrieve components using the tool. The maintainer can also configure the repository, store new components and modify existing components. The current version of CSRTool does not support deletion of components. Deletion deliberately left for future work because

this first version of the tool has not been extensively used yet. More investigation is required to determine the conditions under which deletion of components is required. Since the tool is designed using an object-oriented approach, introducing the deletion facility at a later time should be simple and relatively easy to do. Further, the users are only allowed to query or retrieve components from the repository not to store any new components in the repository. A user who wants to add new components must ask the maintainer to do it. The current version of CSRTool restricts general users to store or modify components in order to prevent from the accidental destruction of the repository.

The detailed architecture of CSRTool is shown in Figure 3.2. CSRTool consists of four main modules. These are *the Storage Manager*, *the Modification Manager*, *the Retrieval Manager*, and *the Type Manager*.

- Storage Manager: Defines and adds new components to the repository.
- Modification Manager: Modifies existing components based on the internal ID.
- Retrieval Manager: Retrieve components using one of two methods: multilevel keyword search and Boolean query search.
- Type Manager: Defines new types, adds new types to the repository, modifies and/or remove existing types. After a new type is created, it invokes a relationship editor to add new relationships between the new type and other existing types.

3.2 Storing Components using CSRTool

The efficiency of the storage mechanism in a repository largely depends on the classification of components that are stored in the repository. Since components exhibit

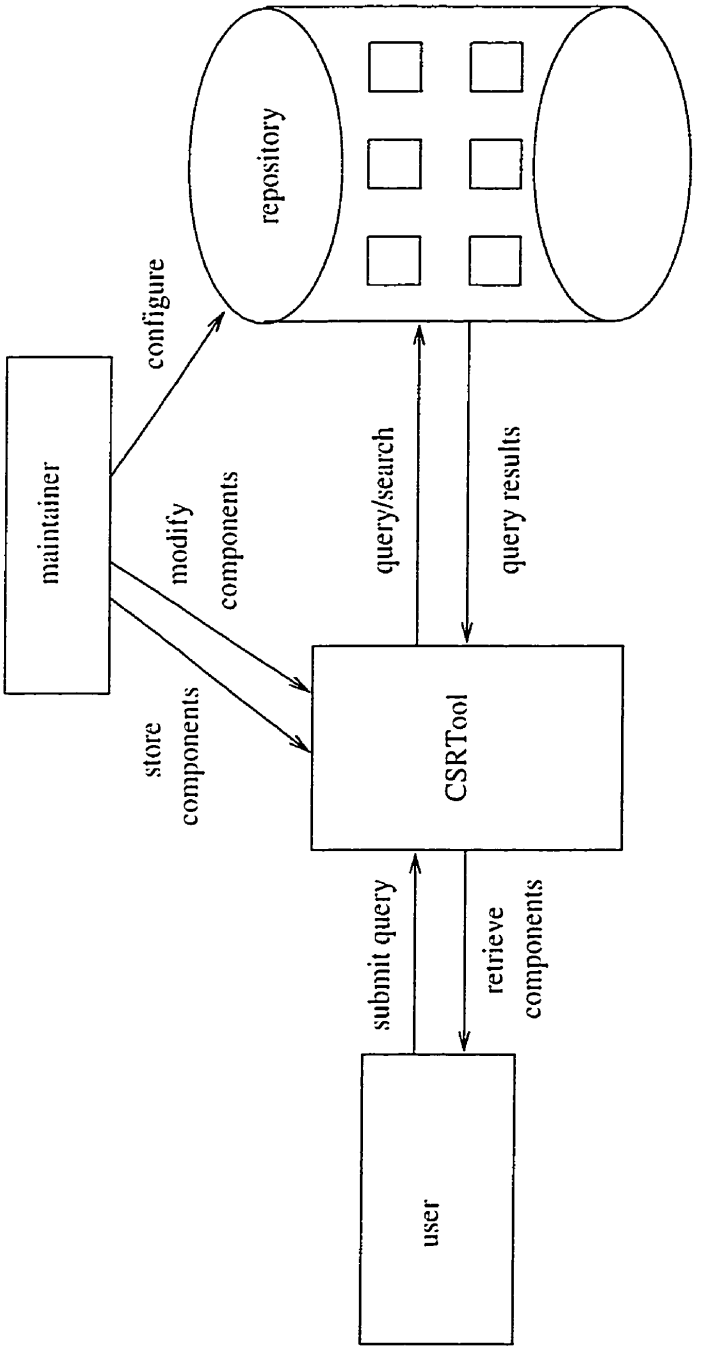

Figure 3.1: The general architecture of CSRTool

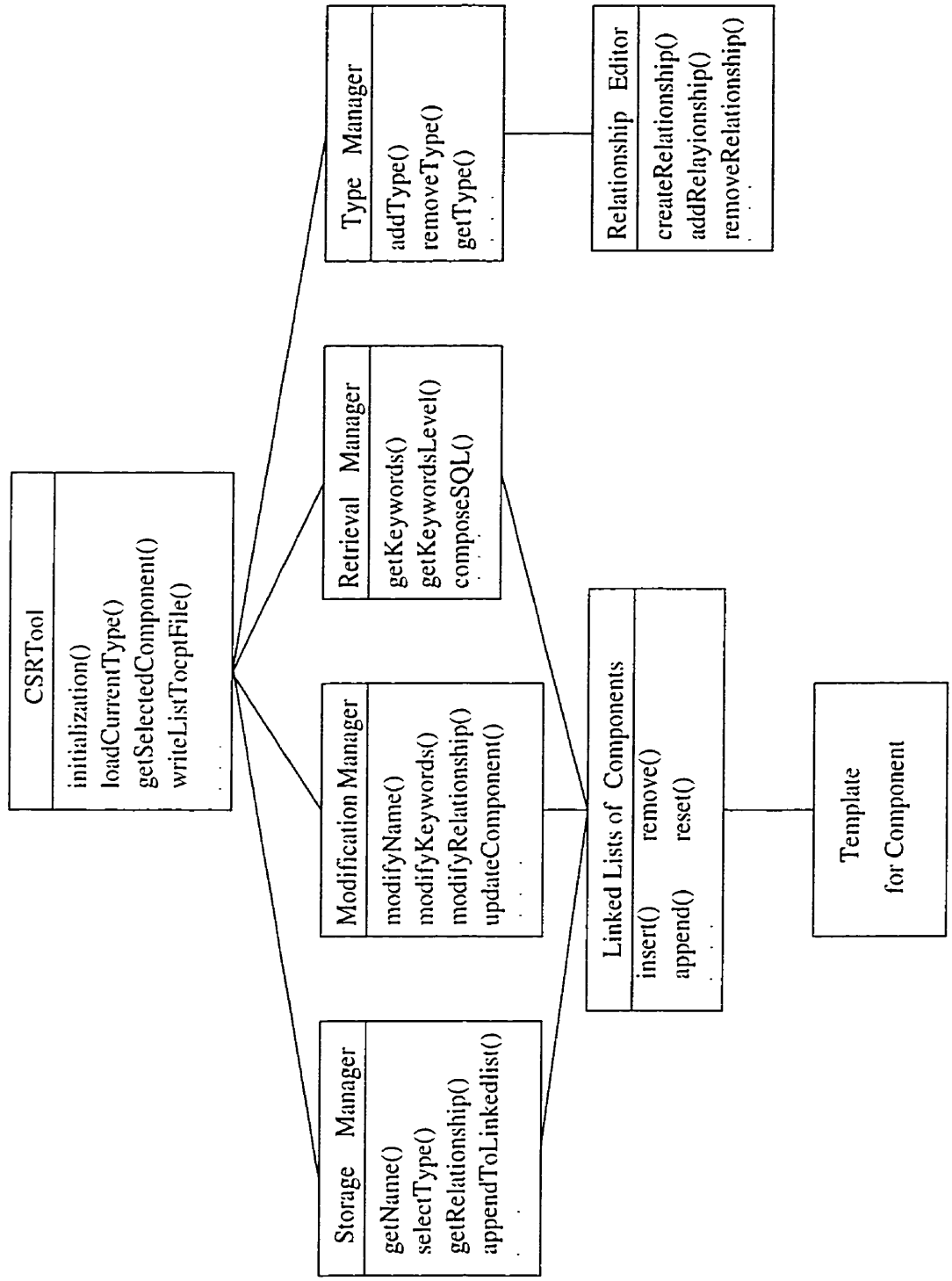

Figure 3.2: The detailed architecture of CSRTool

a set of hierarchical relationships as described in the previous chapter, it is preferable to store components in a hierarchical fashion. Only five types of components are considered in this thesis - *class*, *routine*, *data structure*, *file*, and *documentation*. For convenience of accessing these five types of components, the current version of CSRTool uses linked lists to store the components. There is a separate linked list for each type of components and components of the same type are all linearly connected using the appropriate linked list. The components of the five types are stored in five different linked lists which are named as *Cl-components*, *Ro-components*, *Da-components*, *Fi-components* and *Do-components* respectively. The main reason for choosing the linked list representation is its simplicity in developing the initial version of the CSRTool. The actural repository is stored in file format in the secondary storage. The linked lists corresponding to the various types are dynamically created upon the initialization of system. In addition, the lists can be extended to include any number of components subject to the limitations of the memory size, rather than by programming constraints. Figure 3.3 shows the linked list representation used to store the components. Though the list mechanism is trivial and is well-known, we included its pictorial representation here to show the other pointers that are used along with the list for efficient storage and retrieval.

The node in a linked list of the above form contains a *data* pointer, pointing to a component and a *next* pointer pointing to the next node in the list. Pointers to the first and last node of the list are maintained in two variables called *head* and *tail* respectively. Moreover, the current list pointer is kept in the variable *cur* and the predecessor to current is stored in *pre* for the convenience of traversing the list. The length of the list is stored in a variable *len*. The class implementing the linked list provides the expected list functionality including *insert()*, *remove()*, *append()*, *reset()*, and others that are required for easy manipulation of the list elements.

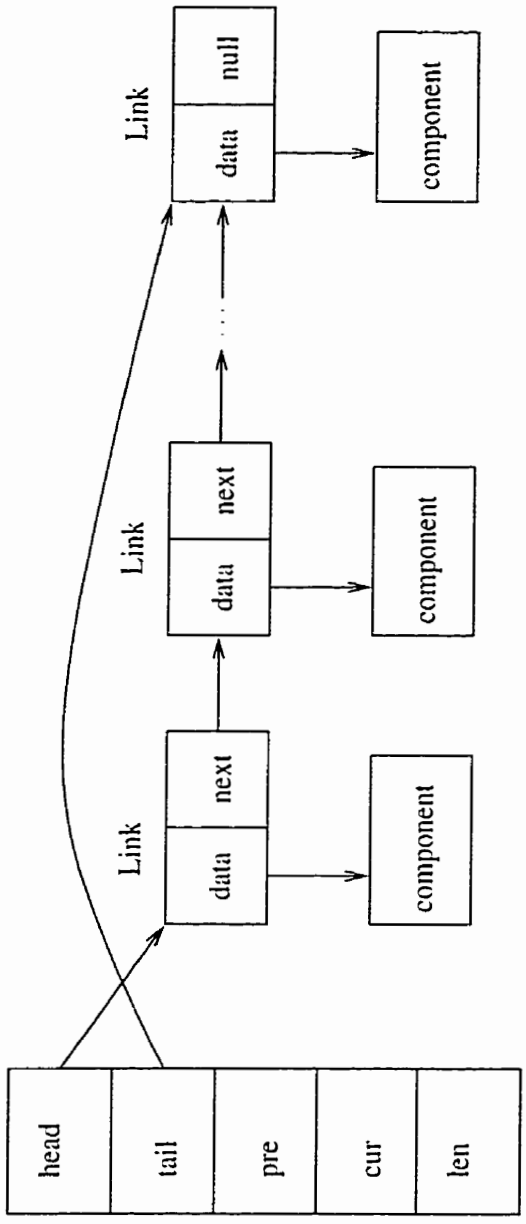

Figure 3.3: The Linked List Representation for Storing Components in CSRTool

The *data* field of the node in the linked list is defined to be of type *Object* in Java, the language in which CSRTool was implemented. Since *Object* in Java represents the root class which every other class in Java inherits, components of any type can be stored in the same linked list. The Unicode Text Format (UTF) was also used to store the information in each node in the list: the purpose being to make the list directly usable on several platforms. All the information stored in these linked lists are loaded into the system during the initialization of the CSRTool. Since the tool has been implemented using Java, it can also be integrated into any database using Java Database Connection (JDBC) with minimal effort.

As new types are added to the repository, new linked lists are created, where each such new list uses the same representation as shown in Figure 3.3. The number of linked lists grow linearly with the number of component types in the repository. Consequently, the performance of the storage and retrieval mechanisms thus depend on the number of types of components stored in the repository.

3.2.1 Component Description

A component in this thesis is described using the following characteristics: the name of the component, a unique internal identifier for the component, the types of the component (notice that a component may belong to more than one type), an abstract description of the component which indicates the functional behavior of the components, the documentation for a component which completely describes the entire component, and a set of relations describing connections to other components related to the current component. The following list describes the characteristics that are used to describe a component in CSRTool:

Identifier This is a unique and immutable identification for each component which

is generated automatically by the system when the component is created. It is used only internally by the tool, but may be passed to other components in the system. Components identified may or may not be part of the current repository. If the other components exist outside the current repository, there must be a data dictionary maintaining the unique identifications of components in various repositories.

Name Each component is assigned a name that is mutable and which may or may not be unique. The name of a component is visible to the user and is selected to indicate the role or functionality of a component in a particular context. Thus, a component of type *file* could be named as "data" when used as input for a program whereas the same component could be called "report" when used as a document explaining a component.

Type The type of a component describes its role in a particular context. As mentioned earlier, a component may belong to more than one type. However, the current version of CSRTool only handles components where each component belongs to a single type. The type of a component can be one of the five types defined previously, or it can be a new type introduced during the usage of the tool. Some changes are anticipated when the tool is later extended to handle components of multiple types.

Keywords A component must have an interface which is described abstractly without indicating the implementation details of the component. This abstract description is supposed to describe the functionality of the component. The purpose of the abstract description is to help identify the component when there is a query for an intended functionality. In the CSRTool, such an abstraction is described in terms of keywords.

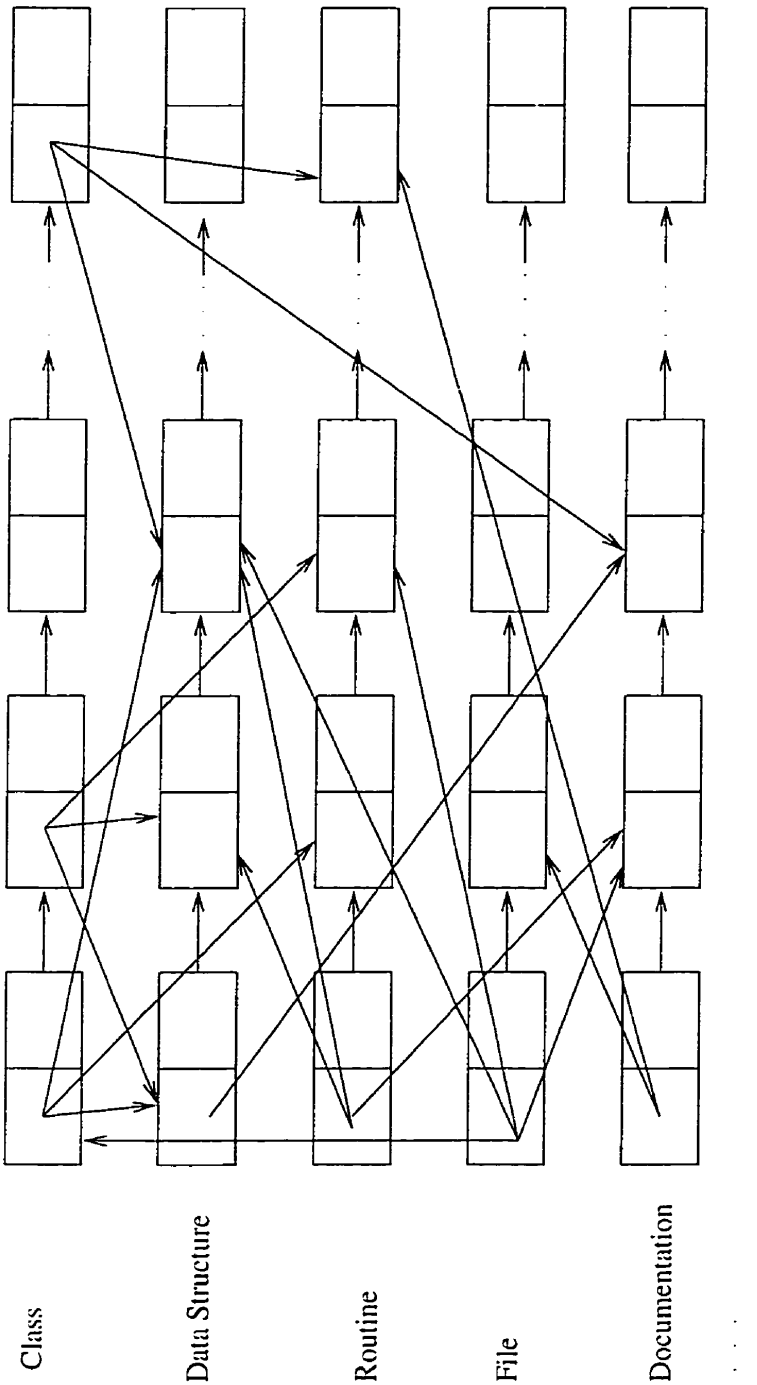

Figure 3.4: Pictorial View of Relationships Among Components

Relations Each component has a set of relationships with other components in the system. These relationships were enumerated in Table 2.1 in Chapter 2. Figure 3.4 shows a pictorial view of the internal links between components. Each directed link from a component X to a component Y represents the information regarding Y such as the unique identifier, name, type, and the name of the relationship between X and Y; this information is stored in X itself. This information can be used to trace the relationships of the component.

Description Every component must have a detailed description in which one can find all information pertaining to the component. This could be a document or a set of documents. For simplicity, the CSRTool reserves a textual description about each component which can be viewed in a separate window.

Date Created This field in repository entry for each component indicates the date at which the component was created; this information is used for maintenance purposes only.

Date Last Modified This field indicates the last date at which the information about the component was modified. Once again, it is kept for maintenance purposes only.

URL Location It is anticipated that the CSRTool will be integrated with the WWW so that components can be searched for and retrieved from other remote systems. The URL where the component is currently stored is therefore maintained as part of the component information.

The graphical user interface to the storage manager of the CSRTool is shown in Figure 3.5. Algorithm 1, given below, indicates the sequence of steps performed in creating and storing a new component:

```
ALGORITHM 1 StoreManager()
Description: This algorithm implements adding a new component to the
repository.
Public StoreManager() {
  /* input name of the component; need not be unique */
  name = get_name ();
  type = select_type (); /* select a type from the types list given */
  /* select keywords in as many levels as possible */
  for i = 1 to 4 do
     if (select_level(i)) keywords(i) = get_keywords();
  /* set the links for related components; a related component can be
     selected in one of the following ways:

     1. If the name and type of the related component is known,
        input the name and type.
     2. If only the name is known, find all the components with this
        name, and select the one desired in this context.
     3. If only the type of the component is known, find all the
        components in this type, and then select the component desired.

     When the desired component is selected, its relationship with the
     new component is selected next (from the permissible
     relationship types  between the two components), and then the link
     corresponding to the related component is added to the current
     component.
  */

  switch (choice) of

    case name_type_known:
        related_component = get_component (rname, rtype);
    case name_known: {
        components_list = select_components_name (rname);
        related_component = select_component (components_list);
```

```
    };
    case type_known: {
        components_list = select_components_type (rtype);
        related_component = select_component (components_list);
    };
  endswitch;
  relationship = select_relationship (type, related_component.type);
  add_link (related_component, relationship);
  description = get_description (); /* could be empty */
  URL = get_URL(); /* could be empty */
  id = generate_newid (); /* a unique ID is generated */
}
END ALGORITHM 1
```

The window *Type* in the GUI shown in Figure 3.5 displays the types of components that are currently available in the repository. The user is expected to choose one of these types. If the user wants to create a component of a new type, he/she must use another manager (to be discussed later).

CSRTool allows a new component to be described by a number of keywords which are used to retrieve the component. Conceptually, these keywords in some way abstractly describe the intended functionality of the new component. These keywords are entered in the boxes to the right of *Level i* (Figure 3.5), $1 \leq i \leq 4$ (the tool currently supports 4 levels only; for the predefined types chosen in this thesis, four levels of keywords found to be more than adequate. However, there is no proof for this claim). The lower the level number, the closer the keyword that actually describes the component. For example, a component of type *document* can be described by the keyword *installation guide* at level 1, and by the keyword *user manual* at level 2. This means that the document is intended to be an installation guide, but when a user searches for the user manual of the software, this component will also be retrieved. A justification for this action comes from the fact that the installation instructions

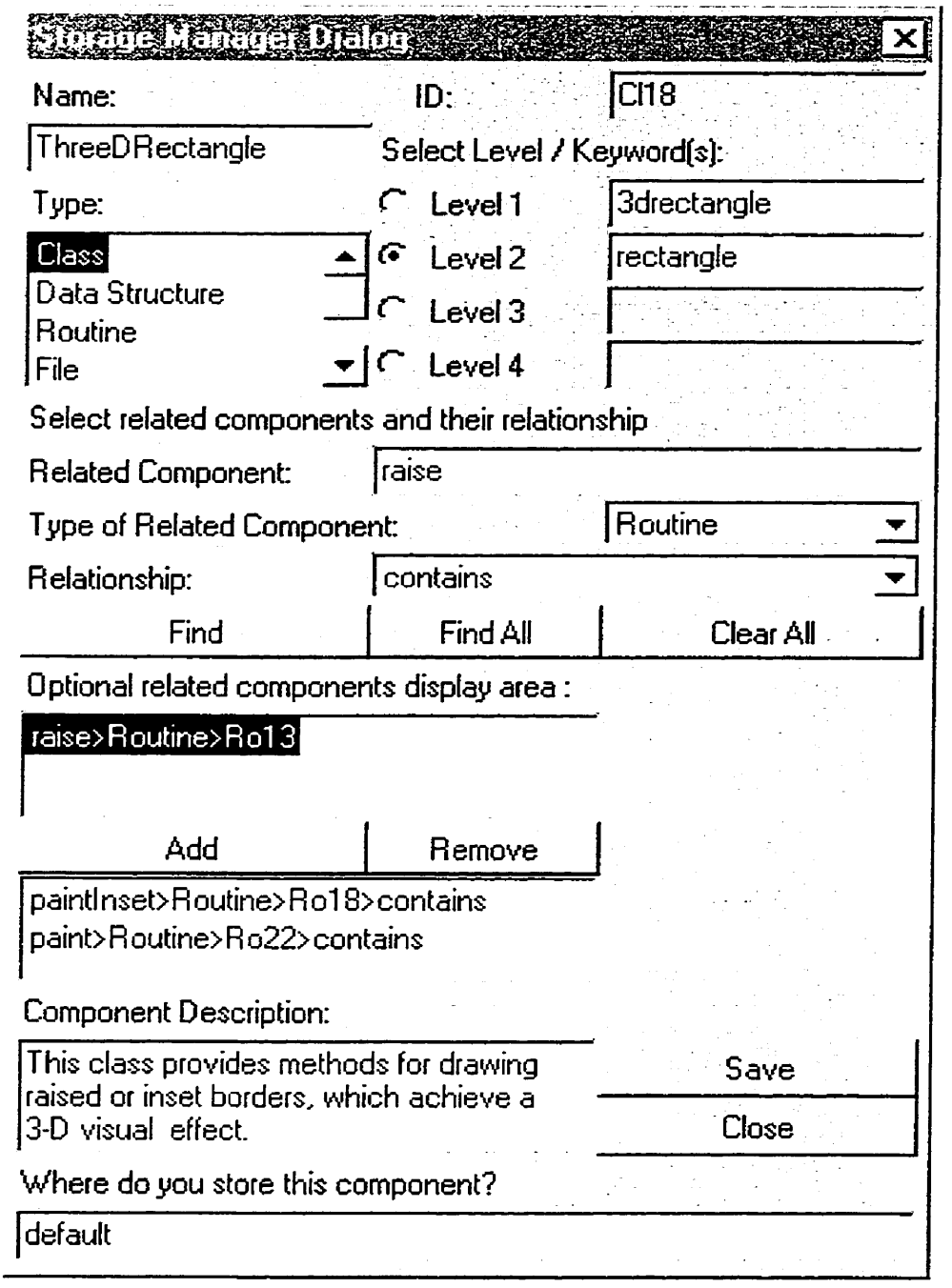

Figure 3.5: The Storage Manager of CSRTool

are, in essence, a part of the user manual. It is also possible for a user to select the level of keywords while searching. This will be discussed in detail in Section 3.3.2.

The set of components that are related to the new component are given in the field *Related Component* (Figure 3.5), one component at a time. Algorithm 1 explains how the user can select a related component when (i) only the name of the related component is known, (ii) only the type of the related component is known, and (iii) both the name and type of the related component are known. In the third category, if the related component does not exist in the repository, the tool will create a new component with the given name, given type, and a unique ID and inform the user that details of this new component must be filled in later. This is not shown in Algorithm 1 because it also involves some work done outside of the algorithm.

After providing a detailed description (textual; can be empty) and a URL (can be empty as well) for the new component, the tool generates and stores a unique ID for this component. The user can then save this new component in the repository.

3.2.2 Modifying a Component

The storage and retrieval mechanisms are controlled by two different modules called the *Storage Manager* and the *Retrieval Manager*. There are also two other related modules, the *Modification Manager* and the *Type Manager*. The former provides a facility by which a user can modify information associated with a component (except its type), and the latter is used to define new types.

Figure 3.6 shows the window of the modification manager. This window is similar to the window used by the storage manager because both access the same set of information for a component. To modify a component, the user must first retrieve the component, and then give its unique internal identifier to the modification manager.

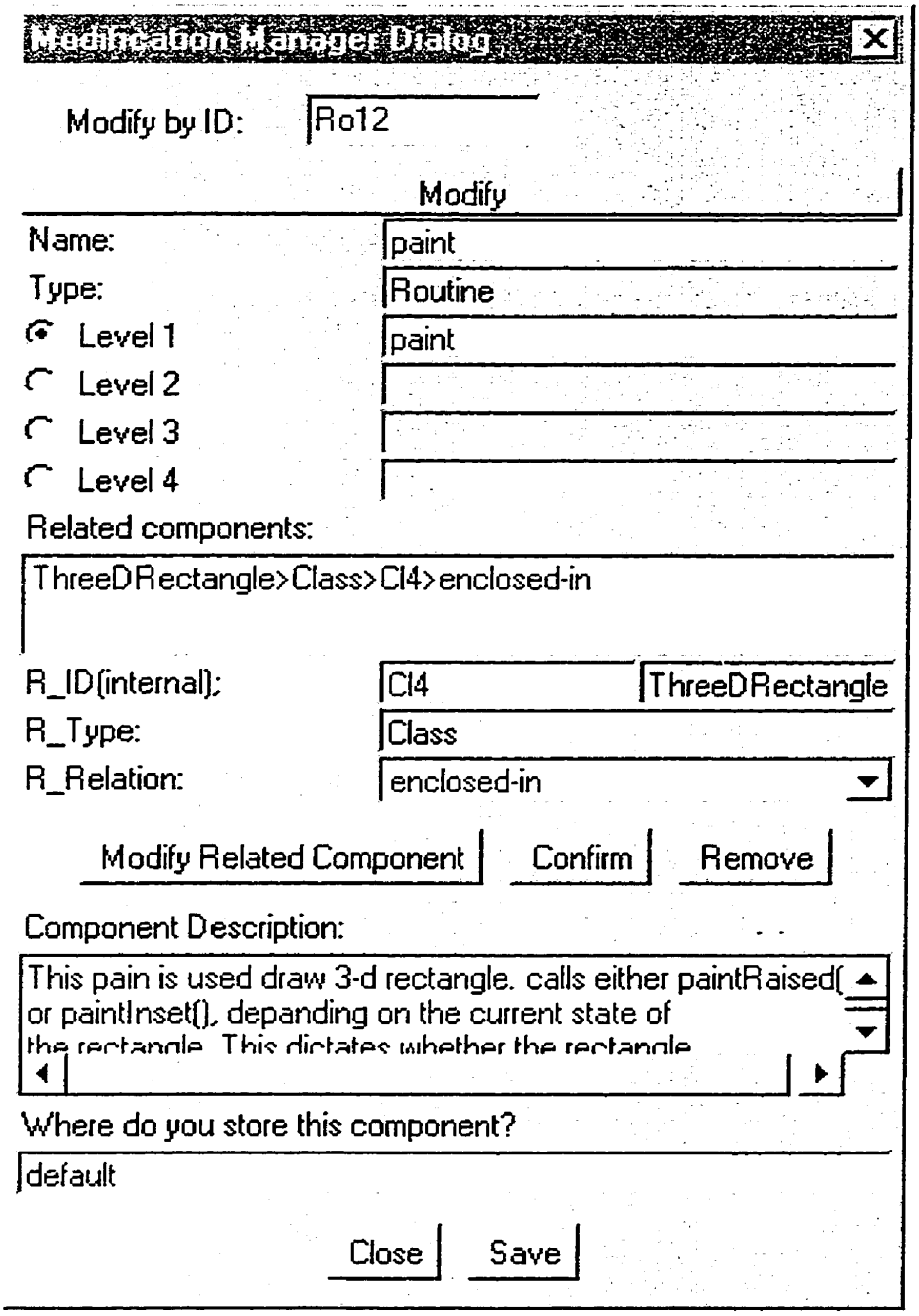

Figure 3.6: The Modification Manager of CSRTool

The modification manager permits the user to modify

1. the name of the component,
2. the keywords and their associated levels which describe the functionality of the component.
3. the relationship between the component being modified and other components in the repository, and
4. the URL in which the component is stored.

Currently the user is not permitted to modify the type of a component since this requires semantic analysis of the component itself, and since changing the type of a component will change the relationships between this component and those that are related to it. The user is still permitted to add or remove existing relationships but these are permitted only within the boundaries defined between the types of components. For example, consider the situation where a component c of type *Class* is being modified. The relationships between c and another component c_1 of type *Class* can include any combination of the relationships defined in the relationships table. Any change in this set of relationships will not affect the relationship between c and other components. On the other hand, if the type of c is changed from *Class* to, say *File*, then every existing relationship between c and other components will be affected.

3.3 Retrieving Components Using CSRTool

The efficiency and usability of the retrieval mechanism in any repository greatly depends on the implementation details of its storage mechanism. Ideally, the retrieval mechanism of the CSRTool should:

1. facilitate searching for components based on several criteria such as keyword search, property-based search, etc. (to promote reuse at all possible levels of abstraction);

2. display sufficient depth of information regarding the characteristics of the retrieved components so that the users of the tool can browse through these details and familiarize themselves about the components and the tool; and

3. facilitate analysis for the intended application.

Four retrieval mechanisms were discussed in Chapter 1: *text-based information retrieval, the faceted-scheme approach, the AI-based semantic network approaches*, and *formal-specifications based approaches*. Of these four, CSRTool uses a modification of the first two approaches. The semantic-based and formal specification-based approaches have not been implemented in CSRTool. Their implementation is left for future work.

As described in Section 3.2, the storage mechanism of CSRTool uses a multiple linked list structure. This structure enables the users to retrieve a small, specific range of components if the component is precisely described or to retrieve a potentially large set of components if the component description is inexact. For example, if the user wants to retrieve any code corresponding to a drawing function, the user can provide the keyword "draw" with the type "Routine". This query will search for components only in the linked list associated with the type "Routine" and retrieve all methods named exactly "draw". On the other hand, a keyword search such as "draw" without any type information will search all linked lists and retrieve all classes, routines, documentations and other types which are related to drawing functionality. This section describes the details of the algorithms that implement the retrieval mechanism of CSRTool and the facilities provided for the users for effective reuse of components.

The retrieval mechanism in CSRTool is a hybrid approach which supports multilevel-keyword search and boolean query search. To improve the efficiency of retrieval, multi-threaded programming is used in the search methods. Section 3.3.1 discusses the design and implementation of multi-threaded search in CSRTool. Algorithms for multilevel-keywords search and boolean query search are described in Sections 3.3.2 and 3.3.3 respectively.

3.3.1 Multi-threaded Search

Multi-tasking is the ability to run more than one program at a time. *Multi-threaded programming* is a conservative extension of the idea of multi-tasking in which a program has the ability to run more than one thread[1] at the same time. CSRTool uses multi-threading extensively, while searching components of multiple types or when type information is not known. In these cases, a separate thread is created for each type to find components in each linked list concurrently, thereby improving the efficiency of searching.

There are two problems associated with the design and implementation of multi-threaded search: *synchronization* and *deadlock*.

Synchronization

A running thread can access any object to which it has reference. If two threads access the same object, and each calls a method that changes the state of the object, then the two threads could end up overwriting each other's modification, leaving the object in an inconsistent state. To solve this problem, we must synchronize access to shared objects in multi-threaded programming. The following synchronization rules, supported by Java, the language in which the tool is implemented, are used:

[1] A thread is usually referred to a computational unit in programming.

- If two or more threads modify an object, declare the methods that carry out the modification as *synchronized.*

- If a thread must wait for the state of an object to change, it should wait inside the object, by calling *wait* inside a synchronized method.

- Whenever a method changes the state of an object, it should call *notify.* That gives any waiting threads a chance to see if circumstances have changed.

Deadlock

Using synchronization can itself introduce a problem. Thread synchronization involve acquiring an exclusive *lock*; i.e., only the thread that currently holds the *lock* can execute. When a program uses more than one lock, *dead lock* may arise. Deadlock occurs when two or more threads are all waiting to acquire a lock that is currently held by one of the other waiting threads in such a way that none can proceed. To prevent deadlock, two measures are taken: (i) using the idea of *semaphore.* In CSRTool, a semaphone for each thread is created, named *thread_flag.* A thread can execute the synchronization method *search* only when it holds semaphore, thereby, avoiding the same thread acquiring extra lock; (ii) keeping the same order for all the threads acquiring locks. By point (ii) the order of waiting is guaranteed to be acyclic and therefore deadlock is precluded.

Algorithm 2 describes the *MultiSearch* thread and its *run* method implemented in CSRTool.

```
ALGORITHM 2 MultiSearch(Type, Keywords, Keywords_level)
Description: A class derived from Thread class to implement
Multi-threaded search.
Type: String; /* the type of retrieved components*/
```

```
Keywords: String; /* the keywords that retrieved components must
satisfy */
Keywords_level: String; /* the levels for keywords searching*/
Output: Retrievedlist: LinkedList /* A linked list which contains
the retrieved components */

Class MultiSearch extends Thread {
    /* constructor of MultiSearch */
    public MultiSearch (String, Type, String Keywords, String
    Keywords_level, LinkedList Retrievedlist)
    /*
       The run method is called whenever MultiSearch is created.
       It is a method of the Thread class, but needs to be
       overridden
    */
    public void run() {
        /*
           run method is a cyclic repeat loop; it checks the
           thread flag, once the thread flag condition is satisfied.
           it calls the search method
        */
        repeat
            /*
              there is a thread array, each thread in the array is
              associated with one type of linked list; if a linked
              list has been searched, it will wait for next search;
              the while loop is not a busy loop
            */
            While ((thread_flag = get_Flag_State()) == done) {
                wait();
            }
            /*
              set done value to thread_flag to avoid repeating the
              same search
```

```
            */
            thread_flag = set_Flag_State(done);
            /* call synchronization search method */
            Search (Type, Keywords, Keywords_level, Retrievedlist);
            /*
              periodtime is a local variable indicating the length
              of time to sleep
            */
            Sleep (periodtime);
        End repeat
    }
}
END ALGORITHM 2
```

Algorithm 3 describes the *Search* method used for synchronization within Algorithm 2.

```
ALGORITHM 3 Search(Type,Keywords,Keywords_level)
Description: This algorithm implements multi-threaded search.
Input: Type: String; /* the type of retrieved components */
Keywords: String; /* the keywords that retrieved components must
satisfy */
Keywords_level: String; /* the levels for keywords searching */
Output: Retrievedlist: LinkedList /* a linked list which contains
the retrieved components */

Public Synchronized void Search(String Type, String Keywords, String
       keywords_level,LinkedList Retrievedlist) {
    /*
      create a temporary array for storing retrieved components
    */
    temp_array = createComponentArray();
    /*
```

```
    retrieve components based on type and level of keywords;
    retrieved components are stored in the temp_array; the
    getSelectedComponents method returns the number of retrieved
    components.
  */
  num_components = getSelectedComponents (Type, Keywords,
                   Keywords_level, temp_array);
  if (num_components != 0) {
      /*
        busyFlag makes sure that the modification of the shared
        structure retrievedlist can not be interrupted; the while
        loop is not a busy wait
      */
      While ((busyflag = getbusyflagstate()) == occupy) {
          wait();
      }
      busyflag = setBusyFlagState(occupy);
      /*
        copy retrieved components from temp_array to the retrieved
        components linked list
      */
      for (i=0;i<num_components;i++){
          Retrievedlist.append(temp_array[i]);
      }
      /*
        set busy flag idle; give the other threads a chance to modify
        shared structure
      */
      busyflag = setBusyFlagState(idle);
  }
  /* call the notify function; the notify function gives the other
     threads a chance for execution
  */
  notify();
```

```
}
END ALGORITHM 3
```

3.3.2 Multilevel Keyword Search

Ideally, the best solution for an automated retrieval method is to input a formal requirements specification and to retrieve desired components from a repository of formally specified components by invoking a theorem prover to determine whether component specifications match the requirements. However, in practice, it is very expensive and difficult to implement such methods. There are two problems that need to be addressed in this context:

- The nature of design problems: software engineers generally begin the design process with an incomplete, inconsistent, and ill-defined set of requirements and understanding of how they should deal with the problems [Bor89].
- A software component's function is often obscure and is not amenable to precise description by a few terms or keywords [Pod93].

Studies have shown that people use a variety of terms to describe the same objects [Fur87]. Therefore, to retrieve components is not just finding an exact match but also includes a close or partial match. The CSRTool has been designed to address these issues by applying multilevel keyword match. The keywords of a component abstractly describe the intended functionality of the component, and are categorized using four levels. The keywords in lower levels more precisely describe the component than the keywords in high levels. By selecting different levels of keywords, a wider range of components can be retrieved, thus implementing exact and partial matches of queries.

Algorithm 4, given below, describes the steps involved in retrieving components using multilevel keyword search.

ALGORITHM 4 MultilevelKeywordsSearch()
Description: This algorithm implement multilevel keyword search. It retrieves components based on the keywords, level of keywords, and type of components. The results are displayed for the user and a retrieved components linked list is created.

```
Public void MultilevelKeywordsSearch() {
    /* Select a type from the types list given*/
    R_Type = getSelectedType();
    /* create a temporary linked list for components*/
    temp_linked_list = createLinkedList();

    Switch (choice) of
        case chose_all:
            /* list all the components in this type */
            retrieved_list = getAllcomponents(R_type);
        case start_search:
            /* select  levels for searching and input the keywords*/
            keywords_level = getKeywordsLevel();
            keywords = getKeywords();
            if (R_Type == AllType) {
             /*
                  invoke multi-threads search. Create thread for each
                  type of component
               */
               thread_array = MultiSearch(R_Type, keywords,
                           keywords_level, temp_linked_list);
               /*
                 create thread flag for each thread, thread flag
                 avoids invoking the same thread repeaedly in a
                 multi-threaded environment
               */
```

```
                thread_flag = createThreadFlag(R_Type);
                /* notify all threads and start threads */
                notify();
                thread_array.start();
            }
            else {
                /*
                  call getSelectedComponents function, retrieving all
                  the components that match the requirements
                */
                num_components = getSelectedComponents(R_Type, keywords,
                                 keywords_level, temp_linked_list);
            }
    End switch
    /*
      select desired components from temp_linked_list and append them
      to the retrieved component linked list
    */
    retrieved_Linked_list = getRetrievedComponents(temp_linked_list);
}
```

END ALGORITHM 4

Figure 3.7 shows the GUI for multilevel keyword search in the CSRTool. The *Chose All* lists all the components of the selected type regardless of the keywords and the levels of keywords. If a user does not know the keywords describing the component to be retrieved, he/she can use this function to browse through and search for the desired components.

3.3.3 Boolean Query Search

It is difficult to answer some questions, such as "How many components in *Class* type have a *draw* Routine as one of their related components?" or "Find all the components

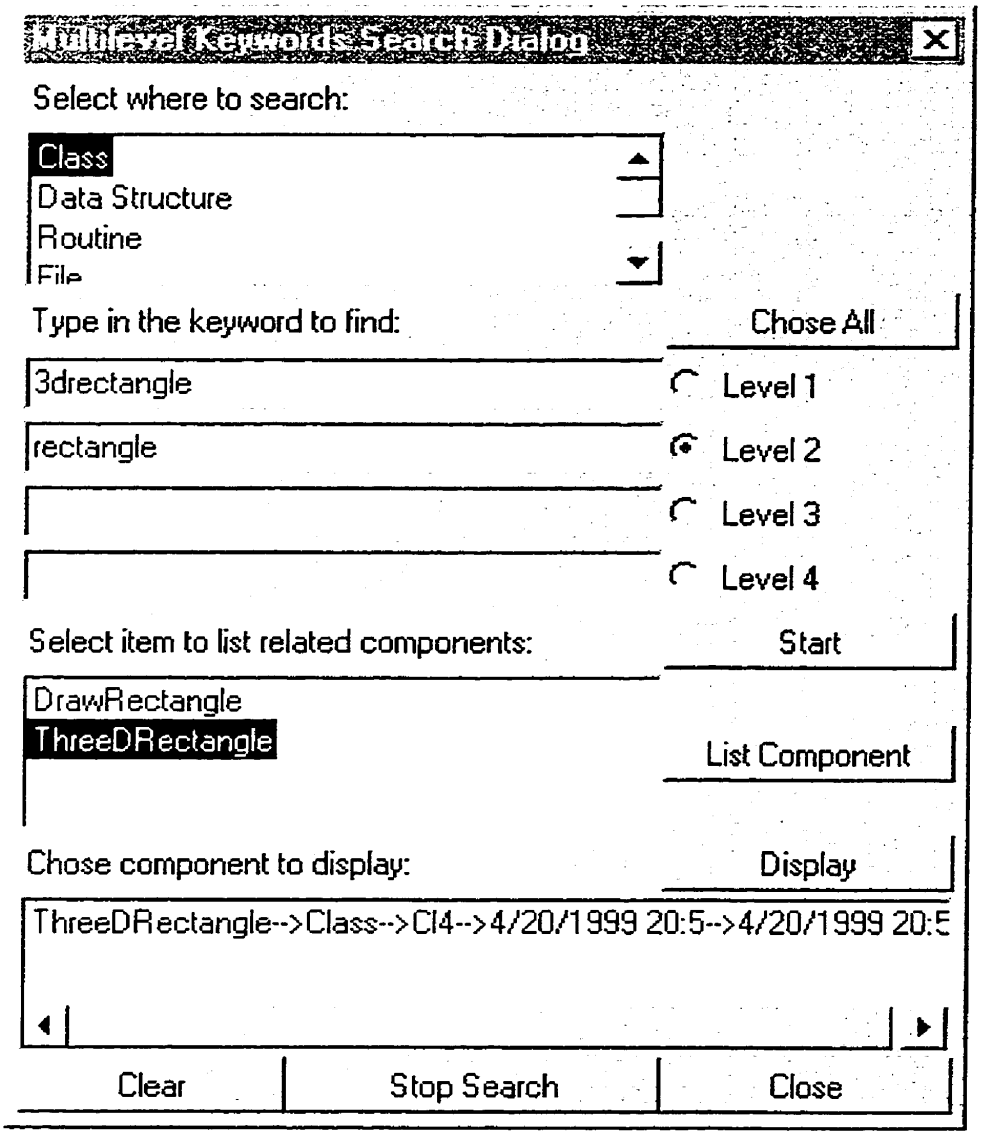

Figure 3.7: Multilevel Keywords Search GUI in CSRTool

of *Document* type such that each has less than three related components.". The reason for the difficulty is due to the keyword-based search. The boolean query search implemented in CSRTool can answer this kind of questions. Table 3.1 shows an example of some query results on the analysis of components in the sample repository; the numbers in the column **Results** indicate the number of components retrieved for the corresponding query.

Query	**Results**
composeSQL(imagemenu. Documentation)	1
composeSQL(draw, Routine. Class)	6
composeSQL(numbergreaterthan, 3, Routine)	27
composeSQL(c_date, 1month, DataStructure)	10
composeSQL(m_date, 2days, File)	5

Table 3.1: Some of the query results based on analysis of components in an example CSRTool repository

Algorithm 5 describes the steps required to retrieve components using boolean query search. Figure 3.8 shows the GUI of the boolean query search of the CSRTool.

```
ALGORITHM 5  BooleanQuerySearch()
Description: This algorithm implements boolean query search. It
retrieves components based on a set of queries. The results are
displayed for the user and a retrieved components linked list is
created.

Public BooleanQuerySearch() {
    /* Select a type from the types list given */
    R_Type = getSelectedType();
```

```
/* create a temporary linked list for retrieved components */
temp_linked_list = createLinkedList();
matchcase = getMatchCaseState();
Switch (choice) of
    case query_name:
        /* input the name of the component */
        Name = getName();
        SQL = composeSQL(Name, R_Type);
    case query_related_components:
        /* input the name of the related component */
        Name = getName():
        /* select type of the related component */
        c_Type = getSelectedType();
        /*
          select logic condition(equal, less than, greater then) and
          input the number preferred
        */
        logicondition = getLogicCondition();
        number = getRelatedComponentsNumber();
        SQL = composeSQL(Name, c_Type, logicondition,number, R_Type);
    case query_date:
        /* select date state: created date or data last modified */
        dateState = getDateState();
        switch (datechoice) of
          case date_between:
            /*
              input start date and end date; the end date by
              default is the current date
            */
            startDate = getStartDate():
            endDate = getEndDate();
            SQL = composeSQL(dateState, startDate, endDate, R_Type);
          case date_month:
            /*
```

```
                input  number of month(s), the range of months is
                from 000 to 999
              */
              months = getMonths();
              SQL = composeSQL(dateState, months, R_Type);
            case date_day:
              /*
                input number of day(s), the range of days is
                from 000 to 999
              */
              days = getDays();
              SQL = composeSQL(dateState, days, R_Type);
          End switch
    End switch
    /*
      select the query's components and append them to the
      retrieved component linked list
    */
    retrived_Linked_list = getRetrivedSQLComponents(SQL);
}
```

END ALGORITHM 5

Algorithm 5 supports boolean query search with three parameters: *Query Name*, *Query Related Components*, and *Query Date*. These three parameters can be combined using the AND or OR operator.

- **Query Name**: When this option is selected, the user inputs the name of component. CSRTool calls the *composeSQL(Name,R_Type)* method to compose a query, which is "Find all components of type *R_Type* such that the components' name is *Name*". Since name need not be unique, CSRTool displays all the components which have that name.

- **Query Related Components**: CSRTool allows the user to retrieve compo-

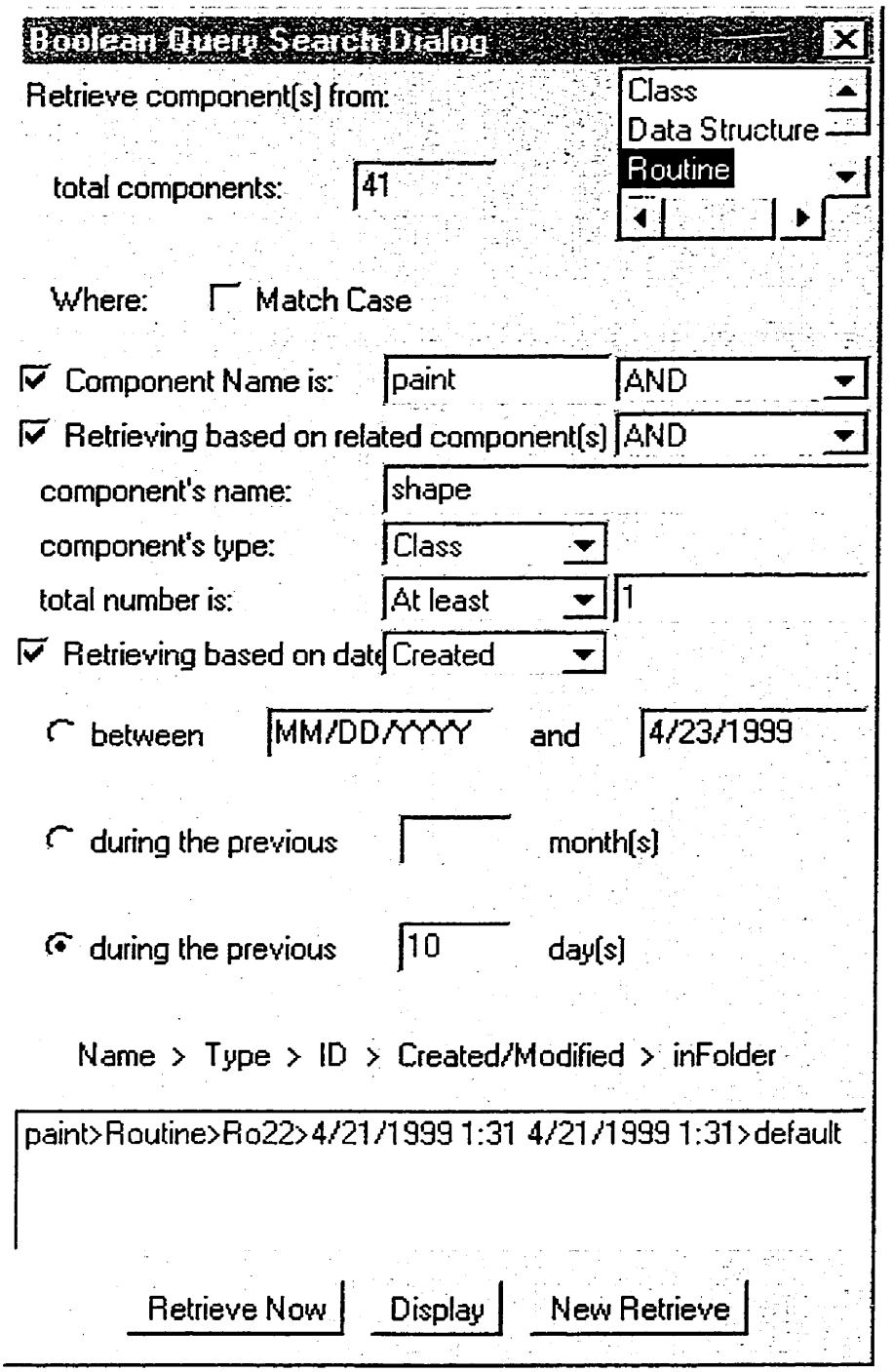

Figure 3.8: Boolean Query Search GUI in CSRTool

nents that are related to a given component based on the name(s) of the related component(s) and /or the type(s) of the related component(s). In addition, the user can also restrict the number of related components that are retrieved based on the above classifications. The tool calls the *composeSQL (Name, c_Type, logi-condition, number, R_Type)* method to compose a query, which means "Find all components of type *R_Type* such that they have a related component' named *Name*, whose type is c_type, and the total number of related components *equals*, or *less than*, or *greater than* given *number*".

The current version of CSRTool does not support queries based on the type of relationship. The reason is that this, again, requires semantic analysis. Querying based on relationships is very useful. For example, we can execute a query like "Find all the components which have an *associates* relationship with component X of Type Y". Implementation of this feature is left for future work.

- **Query Date**: CSRTool supports queries based on a component's *created date* or *last modified date*. From Figure 3.8 one can see that there are three choices for this kind of query. For example, one can pose a query of the form: " Find all components of type *R_Type* such that the component's *created/last modified date* lies between *02/25/1998/* and *03/24/1999*", or " Find all components of type *R_Type* such that the component's *created/last modified date* lies during the previous *6 months*", or " Find all components of type *R_Type* such that the component's *created/last modified date* lies during the last *20 days*".

Boolean query search provides a very flexible means of retrieving component and has proved very convenient for us to quickly browse through, and/or modify the component repository. The combination of multilevel keyword search and boolean query search, and the use of multiple threads improves the run-time efficiency CSRTool.

3.4 Defining New Types

The versatility of a components repository depends on the number of different types of components that the repository supports. The CSRTool currently supports the five types of components - *File*, *Documentation*, *Class*, *Routine* and *Data Structure*. These five types are considered to be primitive for the CSRTool; the relationships of components instantiated from these types have also been predefined in the tool. The definition of a component (as stated in this thesis) permits the user to create and to manipulate a variety of types. Since a component may be composite and may contain any number of other components, one can even treat a package or a database as a component. belonging to the type *package* and *database* respectively. The CSRTool therefore provides a facility to dynamically introduce new types of components in addition to storing and retrieving components of types that are known to the tool already. We call this facility as an incremental refinement of the repository because the configuration of the tool gets modified when new types are introduced.

When a new type is introduced, the storage and retrieval managers must be informed so that they will be able to manipulate components of the new type as well. As described earlier, CSRTool maintains a separate linked list for each type of component. Hence, when a new type is introduced, a new linked list must be created. The relationships between the new type and other existing types must also be defined by the maintainer when the new type is introduced. The table describing the relationships between components types is thus updated and will be used by future references to components of the new type. For illustration, consider the addition of a new type called **GUI-element** and its relationships with the primitive types.

	File	Documentation	Class	Routine	Data Structure	GUI-element
GUI-element	enclosed-in span-over	described-by	contains	contains	contains	contains

Table 3.2: The new relationship between new type "GUI-element" and other types

Algorithms 6 specifies the *type manager* which is responsible for adding new types and their relationships with other types. The GUI for the type manager is shown in Figure 3.9. A relationship editor is invoked to define the new relationships (Algorithm 7); the GUI corresponding to the relationship editor is shown in Figure 3.10.

```
ALGORITHM 6  TypeManager()
Description: This algorithm  implements adding/removing types from
the current type list of CSRTool.

Public TypeManager() {
    /* load current type list */
    currentTypeList = LoadCurrentTypeList();
    /* there are two choices: Adding new type to current type list or
    removing a selected type from the type list */
switch (choice) of
        case add_type: {
            /* input name of new type; needs be unique */
            Type = getType();
            /* add new type to the current type list temporarily for
            later confirmation; if successful; display new type */
            if (addType(Type))
                displayType(Type);
        }
        case remove_type: {
```

```
            /* select type from current type list */
            type = getSelectedType();
            if (removeType(type))
                /*display removed type for confirmation */
                displayType(type);
        }
    end switch
    if (confirmUpdateType()) {
        /* update system type list to maintain consistency */
        updateSystemTypeList(type);
        /* store new updated type list in repository */
        writeUpdateTypetoRepository(type, ''c_type.log'');
        /* invoke relationship editor GUI for defining new
        relationships*/
        invokeRelationshipEditor();
    }
}
```

END ALGORITHM 6

ALGORITHM 7 RelationshipEditor()
Description: This algorithm defines the relationships between the new type and other existing types.

```
Public RelationshipEditor() {
    /* load current type list */
    currentTypeList = LoadCurrentTypeList();
    /* select a pair of types, for example, if the user wants to
    define a relationship between ''GUI-element'' and ''File'', the
    user must select ''GUI-element'' first, then select type ''File''.
    The system automatically composes a GUI_File pair relationship for
    later reference */
    if (selectTypePair()){
        type_relation_pair = getSelectedTypePair();
```

```
        /* serch directory, if type_relation_pair file exists, display it,
        otherwise create one */
        if (getRelationshipFile(type_relation_pair))
            displayRelationship(type_relation_pair);
        else
            createRelationship(type_relation_pair);
    }
    /* there are three choices;
       1. if the user chose add relationship; then select relationship
          from the system relationship list, then add it to the
          current relationship list.
       2. if the user chose remove relation; then select the relationship
          from the current the relationship list, then remove it.
       3. if the user chose modify the predefined system relationship
          list; then input the new relationship, then add it to the
          system relationship list.

    */
    switch (choice) of
        case add_relationship: {
            relation = getSelectedFromRelationship();
            addRelationship(relation,type_relation_pair);
        }
        case remove_relationship: {
            relation = getSelectedFromCurrent();
            removeRelationship(relation,type_relation_pair);
        }
        case modify_system_relationship: {
            relation = inputNewRelation();
            addToSystemRelationship(relation);
        }
    end switch

    if (confirmRelationship()) {
```

```
        /* store the new relationship in the repository for system
        reference */
        writeUpdateRelationshiptoRepository(type_relation_pair);
    }
}

END ALGORITHM 7
```

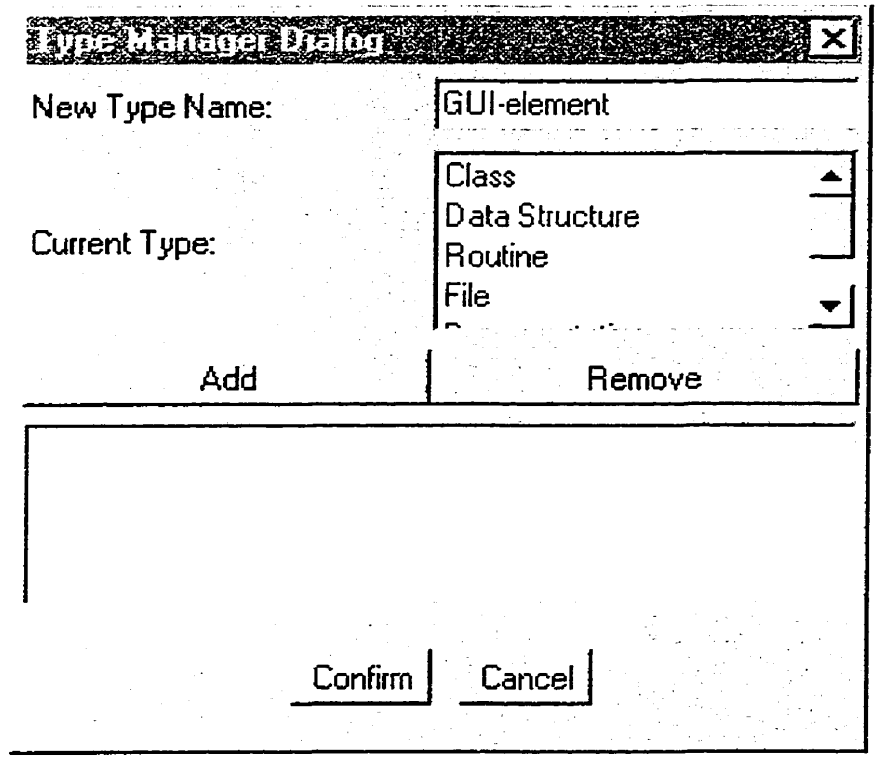

Figure 3.9: The Type Manager GUI of CSRTool

3.5 Implementation

CSRTool has been implemented using the Java programming language under JDK 1.2. Java was chosen for implementation for the following reasons:

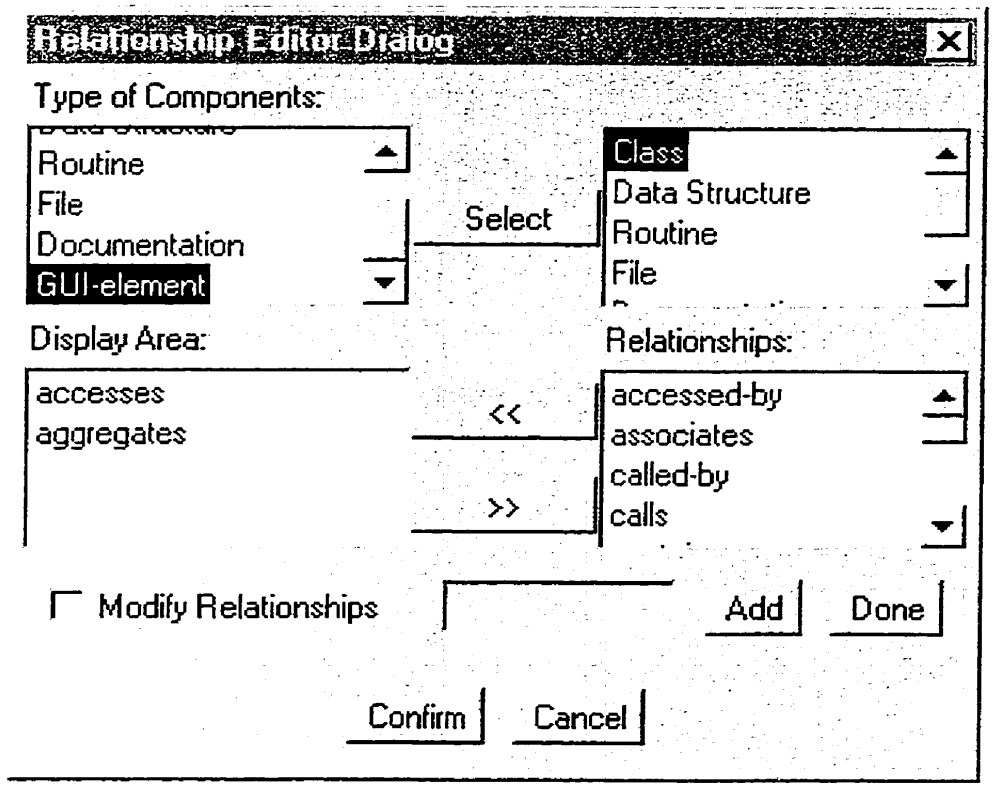

Figure 3.10: The relationship Editor of CSRTool

- Java is platform independent. This makes CSRTool highly portable. The current version of CSRTool runs under Windows NT on PCs, but it can also be used under UNIX; a minimal set of modifications may be required.

- Java supports a rich set of primitives for GUI development. There is a standerized window toolkit called *Alternative Window Toolkit (AWT)* which is an inherent part of the JDK. The AWT contains a variety of GUI primitives which simplified the development of windows for each manager module in the tool. Further, AWT is easy to use and hence modifications to the GUI can be done in relatively short time.

- One of the most important features of Java for the implementation of CSRTool

is its support for multi-threading. As explained in an earlier section, CSRTool uses multiple threads when accessing and manipulating components of several types.

- Another important feature of Java is its support to integrate the CSRTool with the Internet. This will allow integration of the tool with the Internet so that components can be searched for via the Internet as well as added to the repository remotely. The Internet support in Java will help in future extensions of this tool.

- Java also provides garbage collection. This is very important for CSRTool particularly since a number of temporary entries are created during each session of the tool's usage. Without garbage collection, it would be difficult to manage space reclamation in each algorithm. Though garbage collection may decrease performance, it simplifies implementation and thereby increases the reliability of the resulting tool.

- Java is a strongly-typed language. It does not allow arbitrary type conversions, a facility provided by other languages such as C/C++. It is believed that arbitrary type conversions lead to a complicated programs which are difficult to maintain and extend.

- Java does not support pointers. This is a design decision which helps in making the tool more reliable. Problems such as uninitialized pointers and invalid pointer accesses are precluded.

Chapter 4

Conclusions and Continuing Work

Component-based software development (CBSD) is an enhancement of the object-oriented approach to software development. The major distinction between a component in CBSD and an object in OO is that the former can be defined in several levels of abstraction whereas objects are mostly defined at the design and code level. The first challenge in CBSD is not causal the ability to store and retrieve components in a repository. The second challenge is to enable simple plug-and-play combination of components.

The work presented in this thesis is a contribution to the first challenge. A tool implementing such a repository has been implemented in Java, with features to classify, store and retrieve components. By providing precise definitions for classification of components, relationships between components, and the characteristics of components, such a repository can be used in a wide variety of application domains. A hybrid approach for retrieval which supports multilevel keywords search and boolean query search has been presented. The tool has been used to build several prototype repositories; the results are promising.

Some of the limitations of the initial implementation:

- Only storage and retrieval of components has been discussed in the thesis.

- The efficiency of the storage mechanism and that of the retrieval mechanism depend on the classification of components in the repository. If the classification of components is modified, the storage and retrieval mechanisms need to be modified. The classification proposed in this thesis is sufficiently general to be useful in developing software architectures at different levels of abstraction.

- It was assumed implicitly throughout the thesis that the components were already extracted and classified from source files before actually building the repository. It was also assumed that the characteristics of components were correct and consistent. The CSRTool does not perform any validity checking of these characteristics.

- The multilevel keyword retrieval mechanism relies heavily on the keywords selected. Keywords written in different terms may give different results. For example, one can use the keyword "delete" to describe a "Kill" component of "Routine" type while another person might use "remove" as the keyword to describe the same component. It is well known that ensuring consistent use of keywords is very difficult.

- Boolean querying based on the strength of the relationship was not implemented in the current version of CSRTool. Conducting such queries also requires semantic analysis of the components.

The following work items represent possible continuations of this work:

- The current version of the tool uses its own internal structures (i.e. multi-linked lists) for storing components. As the number of components grow, more efficient storage mechanism such as an external database should be considered.

- The tool needs a front end to automatically extract and classify components. Currently, this is done manually. One of the immediate plans for upgrading the tool is to develop this front end.

- Semantic analysis and consistency checks on components have also been reserved for future work. These mechanisms are needed to ensure a formationed plug-and-play ability of components.

Bibliography

[Ale95] P.S.C. Alencar, D.D.Cowan, C.J.P. Lucena, and L.C.M. Nova, "Formal Specification of Reusable Interface Objects", *Proceedings of the ACM Symposium on Software Reliability*, 1995, pp. 88-96.

[Bas98] Len Bass, Paul Clements, and Rick Kazman, "Software Architecture in Practice", *Addison-Wesley*, 1998.

[Bat92] Don Batory and Sean O'Malley, "The Design and Implementation of Hierarchical Software Systems With Reusable Components", *ACM Tranactions on Software Engineering and Methodology*, Vol.1, No. 4, October 1992, pp. 355-398.

[Bat97] Don Batory and Bart J. Geraci, "Validation and Subjectivity in GenVoca Generators", *IEEE Transactions on Software Engineering (Special Issue on Software Reuse)*, Vol.23, No. 2, February 1997, pp. 67-82.

[Bor89] C. L. Borgman, "All users of information retrieval systems are not created equal: An exploration into individual differences", *Information Processing Management*, Vol.25, No. 3, 1989, pp. 237-299.

[Bro97] francois Bronsard, Douglas Bryan, W. (Voytek) Kozaczynski, Edy S. Liongosari, Jim Q. Ning, Asgeir Olafsson, and John W. Wetterstrand, "To-

ward Software Plug-and-Play", *Proceedings of the 1997 Symposium onSoftware Reusability*, Boston, MA, May 17-19,1997, pp. 19-29.

[Cha97] Chao-Tsun Chang, William C. Chu, Chung-Shyan Liu, and Hongji Yang, "A Formal Approach to Software Components Classification and Retrieval", *Proceedings of COMPSAC'97*, Washington D.C., August 11-15, 1997, pp. 264-269.

[Chu98] P. Emerald Chung, Yennun Huang, Shalini Yajnik, Deron Liang, Joanne C. Shih, Chung-Yih Wang, and Yi-Min Wang, "SCOM and CORBA side by Side, Step by Step, and Layer by Layer", *The C++ Report*, January 1998.

[COM94] MicroSoft Inc., "The Component Object Model: Techinical Overview", *Adapted from Dr. Dobbs Journal*, December 1994. http://www.microsoft.com/com/wpaper/Com_modl.asp

[COR95] The Object Management Gruup, "The Common Object Request Broker: Architecture and Specification", http://www.omg.org/corba/corbaiiop.html

[Del97] Chrysanthos Dellarocas, "The SYNTHESIS Environment for Component-Based Software Development", *The 8th International Workshop on Software Technology and Engineering Practice(STEP'97)*, London, UK, July 14-18, 1997.

[Fur87] G.W. Furnas, T.K. Gomez, and S. T. Dumais, "The Vocabulary Problem in Human-system Communications", *Communications, ACM*, November 1987, pp. 964-971.

[Gra97] Mark Grand, "JAVA Language Reference, 2nd Edition", *O'REILLY*, January 1997.

[Hen96] Scott Henninger, "Supporting the Construction and Evolution of Component Repositories", *Proceedings of 18th International Conference on Software Engineering*, March 25-29, 1996 Berlin, Germany, pp. 279-288.

[Hen97] Scott Henninger, "An Evolution Approach to Constructing Effective Software Reuse Repositories", *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 2, April 1997, pp. 111-140.

[Jab98] Khaled Jaber, David Rine and Nader Nada, "Using Adapters at Variation Points in Component-based Software Development: A Case Study", *Proceedings of the 1998 European Reuse Workshop*, Madrid, Spain.

[Kaz97] Rick Kazman, Paul Clements, Len Bass, "Classifying Architectural Elements as a Foundation for Mechanism Matching", *Proceedings of COMPSAC'97*, Washington D. C., August 11-15, 1997, pp. 14-17.

[Kle96] Steve Kleiman, Devang Shah, and Bart Smaalders, "Programming with Threads", *Sunsoft Press/Prentice-Hall*, 1996.

[Maa91] Y.S. Maarek, D.M. Berry and G.E. Kaiser, "An Information Retrieval Approach for Automatically Constructing Software Libraries", *IEEE Transactions on Software Engineering.*, Vol. 17, No. 8, August 1991, pp. 800-813.

[Mey88] Bertrand Meyer, "Object-oriented Software Construction", *Prentice Hall*, 1988.

[Mil94] A. Mili, R. Mili, and R. Mittermeir, "Storing and Retrieving Software Components: A Refinement Based System", *16th International Conference On Software Engineering* ,may 16-24, 1994, sorrento, Italy, pp. 91-100.

[Mil97] ,R. Mili, A. Mili, and R. Mittermeir, "Storing and Retrieving Software Components: A Refinement Based System", *IEEE Transactions on Software Engineering and Methodology*, Vol.23, No. 7, July 1997, pp. 445-460.

[Ost92] Eduardo Ostertag, James Hendler, Ruben Prieto Diaz, and Christine Braun, "Computing similarity in a Reuse Library System: An AI-Based Approach", *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 3, July 1992, pp. 205-228.

[Pfi96] Cuno Pfister and Clemens Szyperski, "Why Objects Are Not Enough?", *Proceedings of First International Component Users Conference(CUC'96)*,Munich, Germany, 15-19 July 1996.

[Pod93] Andy Podguski and Lynn Pierce, "Retrieving Reusable Software by Sampling Behavior", *ACM Transactions on Software Engineering and Methodology*. Vol. 2. No. 3, July 1993, pp. 286-303.

[Pri91] R. Prieto-Diaz, "Implementing Faceted Classification for Software Reuse", *Communication of the ACM*, Vol. 34, No. 5, May 1991, pp. 89-97.

[Ros98] Rational Software Cor., "Rational Rose'98 , User Manu", *Rational Software Corporation*, 1998.

[Rum91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen "Object-Oriented Modeling and Design", *Prentice-Hall, Inc*, 1991.

[Sha96] Mary Shaw, "Patterns for Software Architectures", *Pattern Languages for Programming*, Addison-Wesley, April 1995, pp. 453-462.

[Sha97] Mary Shaw and Paul Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems", *Proceedings of COMPSAC'97*, Washington D.C., August 11-15, 1997, pp. 6-13.

[Voa98] Jeffrey M. Voas, "Certifying Off-the-Shelf Software Components", *IEEE Computer*, June 1998, pp. 53-59.

[Vog98] Andreas Vogel and Keith Duddy, "JAVA Programming with CORBA, Second Edition", *John Wiley & Sons, Inc*, 1998.

[Wol97] Alexander L. Wolf, "Succeedings of the Second International Software Architecture Workshop", *Software Engineering Notes, ACM SIGSOFT*, Vol. 22, No. 1, January 1997, pp. 42-56.

[Zar95] Amy M. Zaremski and Jeanette M. Wing, "Signature Matching: a Tool for Using Software Libraries", *ACM Transactions on Software Engineering and Methodology*, Vol. 14, No. 2, April 1995 , pp. 146-170.

[Zar97] Amy M. Zareski and Jeanette M. Wing, "Specification Mathing of Software Components", *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 4, Oct. 1997, pp. 333-369.