A Case-Based approach to Software Component Retrieval

Carmen Fernández-Chamizo, Luis Hernández-Yáñez, Pedro A. González-Calero and Alvaro Urech Baqué

Dep. Informática - Fac. Física - Universidad Complutense
28040 Madrid, SPAIN
34-1-3944381 (Phone); 34-1-3944687 (FAX); cfernan@dia.ucm.es

Abstract

A major problem concerning the reusability of software is the retrieval of software components. Different approaches, ranging from automatic indexing methods to knowledge-based systems, have been followed to solve this problem.

In this paper we present three prototypes which implement different methods of component indexing. From these prototypes we propose a hybrid system that uses CBR as primary approach but taking advantage of IR methods to facilitate the access to the adequate components.

1. Introduction

Software reuse is widely believed to be one of the most promising technologies for improving software quality and productivity (Biggerstaff & Richter 1987).

Work on reusability has followed several approaches. As Krueger (Krueger 1992) says, we can find very different types and levels of reusability. He divides the different approaches to software reuse into eight categories: high-level languages, design and code scavenging, source code componentes, software schemas, application generators, very high-level languages, transformational systems and software architectures. These approaches rely on reuse techniques that range from abstractions at a very low level, such as assembly language patterns, to very high level abstractions, as software designs. However, there are many technical and non-technical problems to be solved before widespread software reuse becomes a reality. One of the problems is the one concerned with the classification, storage and retrieval of reusable components. Our current work is aimed at the solution of this problem.

If a reusability library is to be successful, it must be structured to help the reuser to find software components of interest, browse through related components to locate the component that best suits the current needs, and build a custom-tailored software component usable in the software system being developed.

The user should be able to find possible components with only an approximate idea of the component's function, without requiring to write a detailed formal specification.

2. Software Component Retrieval

One of the fundamental issues in the reusability of software is the retrieval of software components. In order to reuse a software component, we have to be able to retrieve it from a component base in an easy and efficient way. Therefore, the software component retrieval focuses in two major problem areas: the classification of the software components and the retrieval method. Both problems are interrelated, because the way the components are organized influences the retrieval method to be used.

Traditional approaches to software retrieval fall into two complementary categories: high-level classification techniques, which emphasize retrieval by software category, and low-level cross-reference tools, which facilitate various kinds of browsing at the code level. We

This work is supported by the Spanish Comittee of Science & Technology (CICYT, TIC92-0058)

are primarily concerned with high-level techniques but low-level tools are also needed to establish some important relationships between components at the code level.

The high-level classification of the software components can follow basically two approaches. We can extract the classification information from the component itself (code, documentation). Or, on the other hand, we can implement a classification scheme using information about the components that lies outside of them. That is, we can provide for each component some additional information for the classification scheme to base on; this information could be obtained, for example, from an analysis of the domain. Most of the systems based on the first approach use automatic indexing methods, while the systems based on the second approach are usually "knowledge-based systems".

2.1. Automatic indexing approach

Due to the increasing size of natural language descriptions of software components in recent libraries, information retrieval (IR) techniques based on statistical methods (Salton 1986) are becoming more usual in component retrieval. This approach extracts information from the natural-language documentation of the software components. It doesn't use any semantic knowledge and it doesn't intend to understand the documentation. The goal of this approach is to characterize each component by a set of indices that are automatically extracted from its natural language documentation.

Maarek (Maarek et al. 1991) identifies three requirements that should be fulfilled by IR techniques in the software domain: allow multiple word retrieval, select only key indices, and achieve high precision.

The system proposed in (Frakes and Nejmeh 1987) uses an existing IR system, CATALOG, for storing and retrieving C software components. Each component is characterized by a set of single- term indices that are automatically extracted from the natural- language headers of C programs.

The atomic indexing unit in the Guru system (Maarek et al. 1991) is the lexical affinity (LA). An LA between two units of language stands for a correlation of their common appearance. A set of LA-based indices (or profile) is built for each document by performing a statistical analysis of the distribution of words not only in the document, but also in the corpus formed by the set of all the documents in the repository. LA-based profiles are automatically built for each component description. Those LAs which have a high resolving power in the document with respect to the corpus are selected as indices. In Guru, all software components are classified, stored, compared, and retrieved according to these indices. They can even be organized into hierarchies for browsing using clustering techniques. At the retrieval stage, the user can specify a query in free-style natural language, which is indexed using the same indexing technique. The set of LAs extracted from the query directs the repository search. Various ranking measures can then be used by the Guru system to select the best candidate for a particular query. This system has been applied to the UNIX command set and also to an object-oriented class library (Helm & Maarek 1991).

As the information provided by IR tools is derived automatically, this approach presents advantages in cost, transportability and scalability. Statistical methods, however, can't substitute for meaning.

2.2. Knowledge-Based approach

There is a growing interest in the potential contributions of artificial intelligence to software engineering (Arango 1988). Development of knowledge-based tools for software reusability is one of the most promising research topics in this area.

The key feature of this approach is that it draws semantic information about software components from a human expert. Knowledge-based systems are often very sophisticated. As a tradeoff, they require domain analysis and a great deal of pre-encoded, manually provided semantic information.

Prieto-Díaz (Prieto-Díaz & Freeman 1987) created a classification scheme based on the library science. He proposes a set of six facets: three related to the functionality of the component and three related to its environment. The different values a facet can have are called terms. These terms are structured around certain supertypes that represent organizing concepts. Conceptual distances between terms are assigned by the user. To classify a component, a value for each facet, a term, must be given, so each component is characterized by a six-tuple of terms. The search for a reusable component is accomplished by entering a query with six terms. The set of terms is finite, but a thesaurus is provided to help making the query. The conceptual graph that organizes domain concepts represents manually encoded knowledge about the domain.

The system proposed in (Wood and Sommerville 1988) uses Conceptual Dependency (Schank 1972) to

represent knowledge about software components. They define the "component descriptor frames" in order to represent the function performed by the component and the objects manipulated by the function. After analyzing several application domains, they identified from 10 to 25 basic functions (primitive actions) for software. There is one basic function for each classification of conceptually similar verbs, that is, verbs that describe semantically similar software functions. Also required is a classification of the objects manipulated by software components, into classes or "nominals" that represent conceptually similar objects. The interface to the system is forms-based.

Embley and Woodfield (Embley & Woodfield 1987) define a knowledge structure for a software library consisting of abstract data types (ADTs). This knowledge structure supports different relations among ADTs. The ADTs in the library include natural language descriptions and keywords to assist in finding and browsing activities. Relationships in the ADT library can be automatic (relationships implied by ADT information contained in the library) or user-defined (explicitly provided by the user). Defined relationships for automatically determining closeness, generalizations, specializations, generics, and aggregations implicitly impose a useful structure on the library. In addition, library administrators may also define their own generalizations, specializations, and classifications to further aid the user. The "find" operation, implemented in the prototype, allows several means by which an ADT can be located: by name, by general descriptions in natural-language, by keywords or by operation descriptions. Closeness between natural-language descriptions is determined by evaluating the number and frequency of content words found in both the given and candidate descriptions.

The LaSSIE system (Devanbu et al. 1991) embodies a frame-based knowledge representation. Software components are described in terms of the operations that they perform. Each of these actions is described by giving its actor, object, recipient, agent, environment, and so on. These relationships are coded in a specialized knowledge representation system which classifies them into a conceptual hierarchy. The four principal objects types are OBJECT, ACTION, DOER, and STATE. Nodes below DOER and OBJECT represent the architectural component of the system. Nodes below ACTION represent the system's functional component. The relationship between the two system components is captured by various slot-filler relationships between ACTIONs, OBJECTs and DOERs. This taxonomic hierarchy ensures that components are properly organized and categorized. The taxonomy can also be useful in query formulation and reformulation. When querying the database, if there are no answers or if there are too many answers, the hierarchy can be used to specialize, generalize, or look for alternatives for an appropiate portion of the query; modify this portion and query again. LaSSIE incorporates a natural-language interface which uses a list of compatibility tuples to parse the input. Compatibility tuples, which indicate plausible associations among objects, are obtained from the frame-like knowledge representation mechanism used by LaSSIE.

All these knowledge-based systems have several things in common. They all represent the knowledge in frame-like structures with similar sets of slot-fillers. They all organize the knowledge around the functions performed by the components, although some of them also include frame representations of the objects involved. In every single case, the characterization of the components with slot-fillers is done manually, following a pre-established model of the domain.

Another approach is undertaken in CODEFINDER (Henninger 1991). This system faces a slightly different goal. It explores the problem of finding program examples which are relevant to a design task. CODEFINDER uses an associative spreading activation method for locating software objects.

Curtis (Curtis 1989) has analyzed different software indexing methods used by knowledge-based systems. He postulates that the effective use of a reusable library will require an indexing scheme similar to the knowledge structures built into most programmers working in an application area. The challenge of such indexing schemes is that the organization of a programmer's knowledge will change with increased experience.

3. Previous work

We have developed three prototypes: ARGOS, SARC and PROSA. ARGOS is an IR system which helps UNIX users find out the commands they need. The SARC system implements a component classification mechanism based on facets. The PROSA system is a CBR system for Program Synthesis which indexes software components in a Dynamic Knowledge Base. These prototypes are described below.

3.1. ARGOS: An IR assistant for UNIX

ARGOS (Buenaga et al. 1992) is an on line intelligent assistant for UNIX based on IR techniques, user modeling and hypertext. The goal of this system is to obtain information about the UNIX command (or commands) that implements an operation described by the user using natural-language. The UNIX manual text files constitute the document database. ARGOS uses the method proposed in (Maarek 1991) to characterize each document by detecting the lexical affinities (LAs).

The user will normally use ARGOS to solve a particular problem, but he or she will need to make several queries to find the solution. To display the information on screen, the system takes into account not only the last query, but the whole set of previous queries relative to the user's problem. There is a button for the user to click on when the documentation displayed is irrelevant to his or her problem. The system will use this information about irrelevance when doing the subsequent searches. This is accomplished by the ARGOS' User Model Module. The user may partially reset the user model to indicate that he or she wants to start a new search on a different subject.

ARGOS uses hypertext techniques as navigating tool through the information. When the user wants to know more about a topic, ARGOS searches the related information in the document database, and displays it. The hypertext links are not fixed. They are generated dynamically, depending on the interest of the user.

The user model tries to guess what are the user's interests in order to modify the subsequent searches or to make corrections to the output of the IR module. The user model is incremental. The information is implicitly obtained by the interaction with the user.

The analysis of the documents in the database is performed only once. The results of the analysis are stored in auxiliary files. In the analysis of each document, the following goals are accomplished:

- Only open class words (nouns, verbs, adjectives and adverbs) are considered. Closed class words (pronouns, prepositions, conjunctions and interjections) are ignored.
- Words are taken into their canonical form, eliminating morphemes and suffixes.
- For each document, the set of words and their frequency of appearance is computed. The set of lexical affinities and their frequency of appearance is also computed.
- Total values of frequency of appearance of the words and LAs in the whole corpora are computed.
- Values for the resolving power of each word and LA are calculated for each document.

3.2. SARC: A faceted system for class retrieval

We have also developed the SARC system (Cervigón & Hernández 1992), a prototype that helps the reusers in the retrieval of software components. The software components are the classes of an object-oriented library and the system lets the user select one of two libraries: Smalltalk or Object Professional. The system incorporates a friendly user interface with multiple windows, mouse support and context-sensitive help.

SARC integrates high-level functional knowledge and low-level knowledge. The high-level functional knowledge is built into the classification scheme, a simplified version of the faceted classification of (Prieto-Díaz 1991) with some modifications due to the differences between object classes and general software components. The components are characterized by facets. For each facet, a component has an associated set of terms (or keywords). In order to find candidate components, the user selects one term for each facet and the system shows a list with all the components that match the required terms, if any.

The system also uses low-level knowledge acquired by a deep analysis of the source code of the components. The faceted classification is constructed by the interaction with expert programmers who characterize the new components they incorporate to the library. This means that the characterization is more or less subjective. For this reason, the system uses an affinity metric based in the code shared between any pair of components (Chidamber & Kemerer 1991). These affinity values let the system associate to each candidate a set of "near" components to be considered by the user. The user establishes the affinity threshold the system uses in this search. SARC uses other metrics in order to compute the reuse complexity of the selected components, sorting them by the complexity values.

Once the system displays the selected components, the user can inspect the code and access a hypertext browser that allows him or her to get more information about the selected and other related components.

Finally, when a new component is added to the library, the system asks the programmer for the terms that

characterize the facets of the new component. In order to help the user in the characterization of the facet that reflects the functionality of the component, the system extracts keywords from the comments of the source code. The user may select any of those keywords, or any other word, as terms.

3.3. PROSA: A CBR system for Program Synthesis

The study of the behavior of human programmers can offer useful guidance for supporting software reuse, because human programmers perform many programming tasks by "reusing" previously acquired mental schemas, rather than by reasoning from first principles (Rich and Waters 1988) (Steier 1991).

Programming is usually taught by example. From a set of training programs, students can abstract some program schemas, or even some general programming techniques. Therefore, we consider that CBR (Riesbeck & Schank 1989) is a natural approach to the software design problem. Following this approach we developed a prototype, the PROSA System (Méndiz et al. 1992), which is described below.

The core of the system is a Dynamic Knowledge Base (DKB)(Fernández-Valmayor and Fernández Chamizo 1992), implemented as a discrimination net of frame-like structures named MOPs (Memory Organization Packages) (Schank 1980). This system was initially developed to model the learning processes in a different application domain (Fernández-Valmayor and Fernández Chamizo 1991).

Initially, we introduced into the DKB two kinds of information: a basic Concept Dictionary and a Case Library. The Concept Dictionary is a hierarchical structure that contains the basic terminology to be used in the application domain. At any time, the user can expand the dictionary (and the Case Library) according to his or her needs. Each concept definition is introduced as a name and a property list (set of attribute-value pairs characterizing the concept). We access the concepts by giving their names or by giving their attribute lists. Access by name is provided by a concept table which links each name with the MOP corresponding to the concept in the DKB.

There are two kinds of entities in the Concept Dictionary: objects and operations.

Objects refer to entities that are manipulated by a program. We use primitive objects like characters, integers, links, etc. and non-primitive objects like linear lists, trees, graphs, etc. The system knows implicitly about the primitive objects. Non-primitive objects are introduced into the system giving their names and a set of attributes. New objects are defined in terms of the objects (primitive or non-primitive) known by the system.

Operations refer to the actions that manipulate the objects. We can distinguish between primitive operations (non-defined operations) and non-primitive operations (stated in terms of primitive or previously defined operations).

The system makes abstractions from the common attributes of various objects or operations. When an abstract concept is obtained in this manner, the system asks the user for a name. This name, if provided by the user, will identify the abstract concept.

Each case stored in the Case Library consists of a problem and its solution. A solution is described at different levels of abstraction (stepwise refinement paradigm). At each level, the solution is expressed in pseudocode as the composition of several subproblem solutions (software components). Therefore, we can represent recursively the refinement tree which corresponds to the problem solution. The final refinement level in this tree can be directly translated to an imperative programming language. Problem descriptions and solutions are also introduced into the system as property lists and they are stored in the DKB as instance MOPs. Problem (and subproblem) solutions are accessible by giving the problem attributes or some of the concepts mentioned in the problem (or subproblem) description.

Cases are indexed in the DKB by the operations they implement. The subproblems obtained in the decomposition of the initial problem are also indexed by the operations they implement. These indexes allow the reuse of subproblem solutions in other problems.

When introducing a new training case, the system attempts to find common attributes between the new case and existing cases resident in the DKB, similar to the abstraction process we described in the Concept Dictionary. From these common attributes the system generates abstraction MOPs which represent generic problems. Common attributes are selected from the problem descriptions (without refinement trees). Therefore, they correspond to abstract problem descriptions instead of abstract solutions.

When a new problem is presented, the sytem searches the DKB for analogous problems, by following a top-down strategy. The major goal of PROSA is the

adaptation of the retrieved solutions to solve the new problems. The problem solving mechanisms used in this adaptation process are out of the scope of this paper, where we are concerned with the software indexing methods.

4. A Hybrid Approach to Software Components Retrieval

The goal of our current work is to develop a tool for indexing and retrieving software components from a library of object classes. In fact, this tool will be part of a wider environment that will embody several tools for helping and teaching the usage of the class library. We intend to take advantage of the results of the three previously developed systems, but taking into account the different requirements of the new system:

- Object-oriented design, unlike classical functional design, bases the modular decomposition of a software system on the classes of objects the system manipulates, not on the functions the system performs (Meyer 1987). Therefore, in a class library, the software components are classes of objects which include the operational blocks (methods) that are applicable to these objects.
- In a class library, the documentation is organized around object classes instead of around functional blocks. ARGOS and PROSA assumed that each software description corresponded to a functional block.
- Due to the inheritance mechanism, many class descriptions are incomplete because the inherited methods are decribed in the ancestor classes.
- The Case Library of software components we used in the PROSA system was specifically designed for it. Therefore, the functional description of each software component was tailored to fit the property list required by the prototype. Now, the goal is to apply the same indexing methods to a commercially available class library.

Load

Declaration

constructor Load(var S : idStream);

Purpose

Load a string dictionary from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.
Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to StringDict.Store. Load allocates heap space for the hash table, then reads a series of strings and data values from the stream and adds them to the dictionary.
If Load fails, the constructor will return False and an error code will be stored in the global variable InitStatus.
The stream registration procedure for a StringDict is StringDictStream.

...

Store

Declaration

procedure Store(var S : idStream);

Purpose

Store a string dictionary in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).
Store writes the current size of the hash table followed by a packed image of all the strings and data values for the dictionary. This image may be read by a later call to Load.
To check for errors that may have ocurred during the Store, call the stream's GetStatus function as shown in the example below.
The stream registration procedure for a StringDict is StringDictStream.

...

Figure 1

As starting point, we have selected two widely available class libraries, Object Professional (Turbo Power Software 1990) and Smalltalk (Digitalk 1986), to experiment with. In this way, we can benefit from the experience acquired using the faceted SARC system, that works with the same class libraries. We are particularly interested in the set of terms selected to characterize each class, and in the low-level knowledge obtained from the deep analysis of the source code (inheritance and client relationships between classes).

The class libraries we have selected are relatively small. They contain around one hundred classes and one thousand methods. The associate documentation contains approximately two thousand pages of text, most of which are natural language descriptions with some examples of use at the source code-level.

The user will describe his or her needs by giving a free-text general description of the class or of the methods he or she requires. The retrieving process constitutes an iterative process of querying the system to retrieve adequate components. This process can be implemented as a formulate-retrieve-reformulate cycle (Devanbu et al. 1991). If the answer to an initial query is unsatisfactory, the user can reformulate the query by using partial descriptions of the retrieved classes, or browse amongst functionally related classes. Another approach would be based on a interactive process where, after the first query, the system interrogates the user to disambiguate the meaning of the query. We consider that the browsing approach is the most adequate approach for our initial prototype.

The key points of the system will be the indexing and retrieving methods.

Indexing methods

We will assign an index not only to the classes but also to every method of each class. Each method will be linked to the owner class and each class will be linked to every one of its methods. This will allow the access by class and the access by method.

We intend to use a hybrid approach (knowledge based and automatic free-text indexing) to characterize the software components. We describe this approach in the context of the Object Professional library. A class description may range from 3 to 55 text pages. A method, however, is usually described in less than one page and its purpose is described with only one line. This figures are not mantained in other libraries, but the proportion is more or less conserved. For example, the object StringDict ("string dictionary") is described in twelve manual pages, including the description of its thirteen methods. Some fragments of two of its method descriptions are shown in Fig. 1.

We have analyzed the purpose description of every method in this library, and also some methods of other different libraries, and we can conclude that the syntax and the vocabulary used in these descriptions is very simple. By performing a domain analysis, we can identify a set of basic functions and objects that allows us to express the functionalities of the method. At the moment, we are characterizing each method by a MOP structure that will be stored in the Dynamic Knowledge Base (DKB) that we used in the PROSA system. Following with the previous example, we will have two MOPs characterizing the two methods of Fig.1.

I-M-LOAD.#12 **isA** M-LOAD

selector : Load
implementor : I-M-StringDict
code : I-M-LOAD-IMPLEMENTATION.#15
origin : I-M-Stream
destination : I-M-StringDict

I-M-STORE.#5 **isA** M-STORE

selector : Store
implementor : I-M-StringDict
code : I-M-STORE-IMPLEMENTATION.#8
origin : I-M-StringDict
destination : I-M-Stream

Every method MOP includes three slots specifying the owner class of that method, *implementor*, the actual implementation of the method, *code*, and the identifier of the method, *selector*. The MOP also includes the slots corresponding to the specific action being described (*origin* and *destination* in this case).

These two MOPS will be linked in the DKB under the corresponding action type, LOAD or STORE. Both actions are placed under the basic action ASSIGN, because they are instances of it.

This process will produce a classification of the methods in the library. A method will usually be classified under only one action, but when a method implements several actions it will be placed under all of them.

MOPs corresponding to functionally related methods will be close in the DKB. For example, there are many other "store methods" in the library. One of them

corresponds to the class SingleList and its associate MOP is:

I-M-STORE.#7 isA M-STORE

selector: Store
implementor: I-M-SingleList
code: I-M-STORE-IMPLEMENTATION.#12
origin: I-M-SingleList
destination: I-M-Stream

This MOP will be placed under the same M-STORE MOP that the MOP representing the previous "store method". These methods, however, were not close in the class hierarchy because they belong to different classes.

An abstraction MOP will be generated from the two MOPs because they have the same filler for the *destination* slot:

M-STORE-TO-STREAM isA M-STORE

destination: I-M-Stream

A class description, as previously stated, can extend to several text pages. Therefore, it would be a very complex task to express all this information as MOPs. We think this is the right place to use statistical indexing methods. So, we will use ARGOS to obtain the lexical affinity based indices of every class in the library. These indices will be used in the first step of the analysis of the user's query. Furthermore, we will create for each class a MOP containing information about its structural description, its relationships with other classes (inheritance, client) and the links to all its methods. The structural description defines a set of values for the class, i.e. a list of other classes needed to construct the current class. For example,

```
StringDict =                          {Dictionary of strings}
    object(Root)
      sdSize : Word;   {Maximum elements in hash pool}
      sdUsed : Word;   {Current elements in hash pool}
      sdPool : HashPoolPtr;     {Pointer to hash pool}
      sdStatus : Word;                {Status variable}
```

The structural description and the relationships with other classes will be obtained from the source code using the low-level tool of the SARC system. MOPs corresponding to classes are stored in the DKB under the MOP OBJECT at the root level of the memory. A fragment of the resulting memory organization is shown in Fig. 2.

Retrieving method

The analysis of the user's query will be made in two steps:

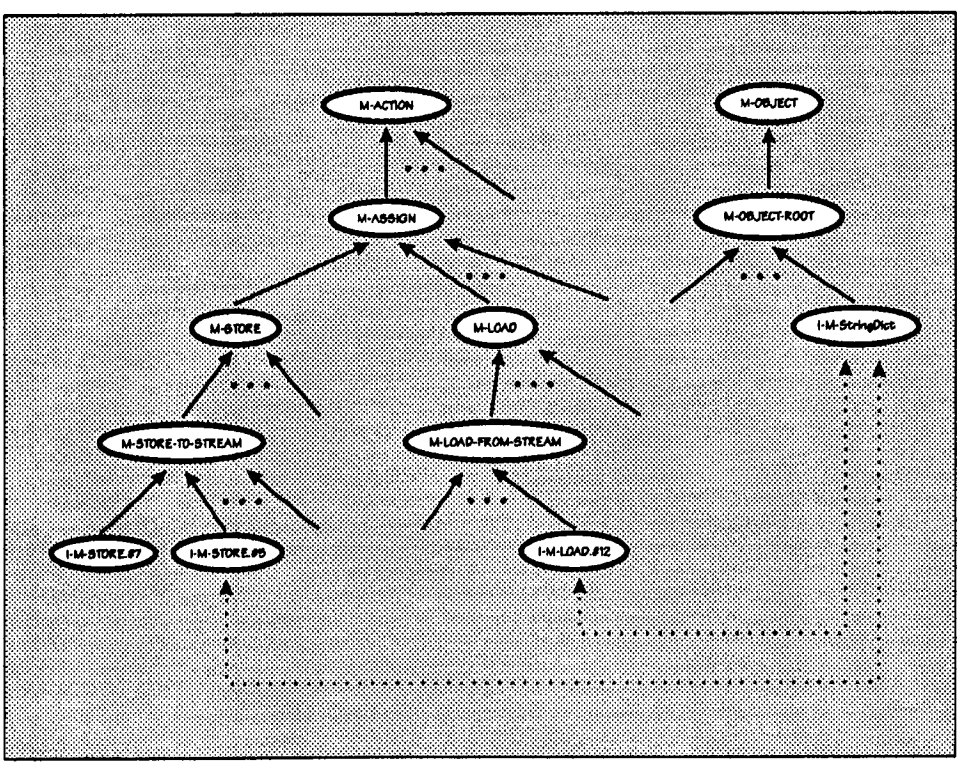

Figure 2

1. The query is indexed by the same statistical method used to index the classes. The profile of lexical affinities detected in the query allows the access to the classes with similar profiles. MOPs corresponding to these classes and all their methods will be activated.

2. The query is analyzed again by an expectation-based parser (Dyer 1983) (Martin 1989) which uses the MOPs activated in the step 1 as expectations. When the parser recognizes the methods or the structural description of a class, this class will be displayed as an answer to the query. A complete matching between the query and the selected class is not necessary, because the class required by the user may not exist in the library. In this case, the user wants to obtain a functionally close class to adapt it. If the parser has recognized several classes, the class with more recognized methods will be displayed. If several classes have the same number of recognized methods, the rate between the functionality required by the user and the global functionality of the class will be used to select the candidate class. For example, if the query was "a dictionary to be used to detect whether a particular word is in the dictionary or not", two classes would be selected, StringDict and StringSet. Both implement the two methods required, but the first one implements another eleven methods and the second one doesn't have any others. So the StringSet will be displayed because it fits better the user requirements.

This parser has to be completed with a browser. This browser will allow to explore the classes functionally related to the class selected by the system. The browser will also allow to explore similar methods belonging to different classes.

The current state of the prototype only supports the indexing methods. We are undertaking the design of the parser and specifying the browser features. We are also recodifying some parts of the prototype because the previous systems were implemented in different languages and machines.

The prototype is being implemented in BIM Prolog under SunOS 4.1 in a SUN Sparc workstation.

5. Conclusions

We have presented a hybrid approach to the indexing and retrieving problem for software components. This approach consists of using CBR techniques to represent knowledge about classes and methods, but simultaneously using statistical indexing methods to facilitate the access to classes.

Automatic indexing methods have proved to be succesful when applied to software components. We don't have any results that predict better results in component retrieval by adding knowledge based methods.

Anyway we believe that CBR approaches can be useful when teaching (or aiding) the user to reuse software components. If some of the components in the library come from past designs, the experience acquired in the designing of such a system can be of use to novices. Even experienced designers, can save time and effort using reusable design templates and exploring rationale from previous designs, instead of finding their way through trial and error prototyping.

As we undertook not only the retrieving task but also the educational task, we need to represent the user's mental model of the domain, and also be able to reason about the functionality of the components. Therefore, in order to develop an environment which helps programmers learn about the library's structure and contents, we have chosen CBR as the primary approach. We expect that this approach will produce, as a side-effect, a better performance in the retrieving of software components. Even in this case, the information provided by IR tools would be useful since it can be derived automatically.

6. References

Arango, G., Baxter, I. & Freeman, P., 1988. *A Framework for Incremental Progress in the Application of Artificial Intelligence to Software Engineering*, ACM Software Eng. Notes, vol.13, no.1, 46-50.

Biggerstaff, T.J. & Richter, C., 1987. *Reusability Framework, Assessment, and Directions*, IEEE Software, vol.4, 2.

Buenaga, M., Fernández, B., Vaquero, A. & Urech, A., 1992. *An On Line Intelligent Assistant for Unix*, Tech. Report DIA 92/16. Department of Computer Science, Complutense University of Madrid.

Cervigón, C., Hernández, L., 1992. *Manual de uso del Sistema de Ayuda para la Recuperación de Clases*. Tech. Report DIA 92/11. Department of Computer Science, Complutense University of Madrid.

Chidamber, S.R. & Kemerer, C.F., 1991. "Towards a Metrics Suite for Object-Oriented Design". *OOPSLA-91*.

Curtis, B., 1989. "Cognitive issues in reusing software artifacts", *Software Reusability, volume II, Applications*

and Experience, (Biggerstaff, T.J. & Perlis, A.J., eds.), ACM Press, Addison-Wesley Publishing Company.

Devanbu, P., Ballard, B.W., Brachman, R.J. & Selfridge, P.G., 1991. "LaSSIE: A Knowledge-Based Software Information System", *Automating Software Design* (Lowry, M.R. & McCartney, R.D., eds.), AAAI Press/ The MIT Press.

Digitalk, 1986. *Smalltalk/V. Tutorial and Programming Handbook.*

Dyer, M., 1983. *In Depth Understanding*. MIT Press.

Embley, D.W. & Woodfield, S.N., 1987. "A Knowledge Structure for Reusing Abstract Data Types", *Proceedings of the Ninth International Conference on Software Engineering*, ACM, 360-368.

Fernández-Valmayor,A. & Fernández Chamizo, C., 1991. "An Educational Application of Memory Organization Models", *Advanced Research on Computers in Education* (R. Lewis & S. Otsuki, eds.). Elsevier Science Publishers B.V., IFIP.

Fernández-Valmayor,A. & Fernández Chamizo, C., 1992. *Educational and Research Utilization of a Dynamic Knowledge Base*, Computers & Education, 18, No. 1-3, 51-61.

Frakes, W. B. & Nejmeh, B. A., 1987. "Software Reuse through Information Retrieval". *Proceedings of the 20th Annual HICSS*, Kona, HI.

Helm, R., Maarek, Y.S., 1991. "Integrating Information Retrieval and Domain Specific Approaches for Browsing and Retrieval in Object-Oriented Class Libraries". *OOPSLA-91*.

Henninger, S., 1991. "Retrieving Software Objects in an Example-Based Programming Environment". *Proceedings of the Fourteenth Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval.*

Krueger, C.W., 1992. *Software Reuse*. ACM Computing Surveys, vol. 24, 2.

Maarek, Y.S. et al., 1991. *An Information Retrieval Approach for Automatically Constructing Software Libraries*. IEEE Transactions on Software Engineering, vol. 17, 8.

Martin, C., 1989. "Case-based parsing". *Inside Cased-Based Reasoning*. Riesbeck, C.K. & Schank, R.C.(eds.) 1989. Hillsdale, N.J. Lawrence Erlbaum Associates.

Méndiz, I., Fernández-Chamizo, C. & Fernández-Valmayor, A., 1992. "Program Synthesis Using Case Based Reasoning", *12 IFIP Congress, Poster Session: Software Development and Maintenance*, Madrid, Sept. 1992.

Meyer, B., 1987. *Reusability: The Case for Object-Oriented Design*. IEEE Software, vol.4, 2.

Prieto-Díaz, R., Freeman, P., 1987. *Classifying Software for Reusability*. IEEE Software, January.

Prieto-Díaz, R., 1991. *Implementing Faceted Classification for Software Reuse*. Communications of the ACM, vol. 34, nº 5.

Rich, C. & Waters, R.C., 1988. *Automatic Programming: Myths and Prospects*, Computer, August 1988, 40-51.

Riesbeck, C.K. & Schank, R.C., 1989. *Inside Cased-Based Reasoning*. Hillsdale, N.J. Lawrence Erlbaum Associates.

Salton, G., 1986. *Another look at Automatic Text-Retrieval Systems*, Communications of the ACM, vol.29, 7.

Schank, R.C., 1972. *Conceptual Dependency: A Theory of Natural Language Understanding*, Cognitive Psychology, 552-631.

Schank, R.C., 1980. *Language and Memory*, Cognitive Science, 4, 243-284.

Steier, D., 1991. "Automating Algorithm Design within a General Architecture for Intelligence", *Automating Software Design* (Lowry, M.R. & McCartney, R.D., eds.), AAAI Press/ The MIT Press.

Turbo Power Software, 1990. *Object Professional User's Manual.*

Wood, M. & Sommerville, I., 1988. *An Information Retrieval System for Software Components*, ACM SIGIR, vol. 22, 3,4.