Automating the Indexing and Retrieval of Reusable Software Components

Hafedh Mili[1], Petko Valtchev[1], Anne-Marie Di-Sciullo[2], Philippe Gabrini[1]

[1] Département d'Informatique, UQAM
[2] Département de Linguistique, UQAM
C.P. 8888, succ. "Centre Ville", Montréal, Québec, Canada, H3C 3P8

Abstract: Developing libraries of reusable components is a key issue in the software reuse field. In this paper, we focus on a set of indexing and retrieval methods devised for the *ClassServer* experimental library tool. We compare string search-based retrieval methods to keyword-based ones. As the main concern in reuse is cost, we argue that the second method, which is widely believed to be far more expensive in development resources than the first one, might be successfully assisted by some (semi-) automatic tools to reduce its actual cost. We thus describe a possible way to automate two of the pre-processing steps, the identification of the domain concepts to form the controlled vocabulary and the extraction of the vocabulary's hierarchical structure. Both methods have been implemented and tested within the *ClassServer* system. Notwithstanding the results of preliminary experiments, we were able to observe the positive effects of our approach and identify possible enhancements for both extraction methods and experimental design.

1 Introduction

Software reuse is seen by many researchers as an important factor in improving software development productivity and software products quality. There are many challenges to software reuse, including managerial practices and organizational structures, in addition to the myriad of technical challenges that need to be addressed [MMM95]. In this paper, we focus on computer support for software component search and retrieval. A wide range of component categorization and searching methods have been proposed, from the simple string search (see e.g. [Mil94]) to faceted classification and retrieval (e.g. [PDF87, Ost92]) to signature matching (see e.g. [ZW93]) to behavioral matching (see e.g. [ZW95, MMM94]) or even [Hal93]. Different methods strike different trade-offs between performance and cost of implementation, including both initial set-up costs, and the cost associated with formulating, executing and refining queries [MMM95]. Retrieval experiments have systematically used *recall* and *precision* measures. However, these measures do not take into account the intended use of the retrieved items, and often assess relevance in a binary, yes/no, fashion. Furthermore, for the case of reusable components where cost is a factor, we argue that the developing costs of the various methods should not be ignored in a fairly set comparison.

The study reported in this paper compares two kinds of indexing and retrieval methods, a full-text retrieval and a multi-faceted classification using a controlled vocabulary. To bring both methods to a level-playing field we tried to automate as much of the pre-processing involved in controlled vocabulary-based methods as possible. In what follows, we describe the approaches we suggest for both the discovery of useful domain concepts and their hierarchical ordering, and the experiments of the possible uses of the resulting controlled vocabulary for retrieval purposes.

The paper starts with a description of our experimental library tool, *ClassServer* (Section 2). Section 3 presents two competing approaches for component indexing and retrieval in *ClassServer*. Section 4 describes our automatic approach to vocabulary constructing and document indexing. Section 5 summarizes the results of the experiments with the automatic tools as well as with the resulting library.

2 The *ClassServer* tool

Our work on supporting development with reusable components centers around a tool kit called *ClassServer* which is described below.

2.1 System architecture

ClassServer integrates a set of tools for representing, classifying, retrieving and navigating through reusable components (see Figure 1). Components consist essentially of object-oriented source code components, occasionally with the accompanying textual documentation. Raw input source files are put through various tools called extractors which extract the relevant pieces of information, and package them into *ClassServer*'s internal representation format for the purposes of representing, classifying, indexing and retrieving components. So far, we have developed extractors for Smalltalk and C++. The information extracted by these tools includes language-defined structures, such as classes , variables , methods , and method parameters for the case of C−−. Figure 1 shows a very schematic

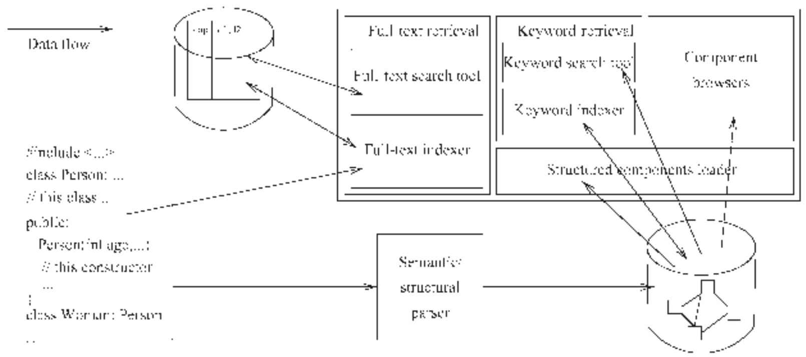


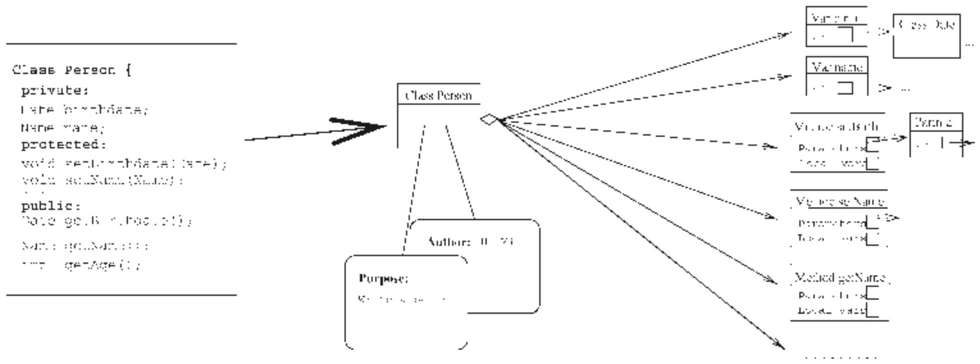
Fig. 1: Overall architecture of *ClassServer*.

view of *ClassServer* which may be seen as a sum of three subsystems whereby the first one supports the required functionalities for full-text retrieval. The other subsystems, i.e., the component browser and the keyword retrieval subsystem, use the structured representation of the components that is extracted by the "semantic/structural parser" (see Figure 1) to support controlled vocabulary retrieval.

2.2 Representing components

Each kind of component is defined by a descriptive template that includes: 1) structural information describing the kind of subcomponents a component may have (e.g. a *class* has *views*, a view has *variables* and *methods*), 2) code, which is a string containing the definition or declaration of the component in the implementing language, and 3) descriptive attributes used for search purposes like an `author` and an `application domain` for a class, a `purpose` for a method, etc. Figure 2 shows parts of the network of components representing a typical C++ class. This particular network includes four kinds of components: classes, variables, methods, and method parameters.

The attributes represent non-structural non-intrinsic properties of software components; they are often derived from non-code information such as documentation, or entered explicitly by the "component library manager" [PD90]. Attributes have two properties of their



```
Class Person {
 private:
 Date birthDate;
 Name name;
 protected:
 void setBirthDate(Date);
 void setName(Name);
 ...
 public:
 Date getBirthDate();
 Name getName();
 int  getAge();
```

Fig. 2: Representing a C++ class with a network of reusable components.

own: 1) `text`, a (natural language) textual description, and 2) `values`, a collection of key words or phrases, taken from a predefined set which is referred to as the vocabulary of the attribute. The text is used mainly by human users and for documentation generation [Mil94]. Multiple values in an attribute are considered to be alternative values (ORed), rather than partial ones (ANDed). For example, for the attribute `Purpose` of a component, several values mean that the component has many uses, and not a single use defined by the conjunction of several terms.

2.3 Example library

Our experiments were carried out over the entire *OSE* library [Dum94] which contains some 200 classes and 2000 methods distributed across some 230 *.h files with, typically, one class per file. For the purposes of supporting plain-text indexing and retrieval, the 230 files were put through the plain text indexing tool, which generated an inverted list of unique word stems. Further, a shell script put the files through a C++ pre-processor before they were input into the C++ extractor. In addition to source code, the library includes a considerable amount of on-line HTML documentation. Because of its good quality and format consistency, we were able to extract individual paragraphs from the documentation and assign them as text values for the `Description` attribute of various components (classes, methods, variables).

3 Component retrieval within *ClassServer*

We have implemented a set of concrete retrieval methods within *ClassServer*. In what follows, we focus on "controlled-vocabulary" versus "full-text" indexing and retrieval.

3.1 Full-text versus controlled-vocabulary indexing

Before enabling a retrieval mechanism, the documents/components in a library should be properly indexed. Filling in the `values` property is referred to as *classification*, *categorization* or *indexing*. Typically, the task is carried out manually: human experts read about the software component, and select key words or phrases that best describe its "content", this assigning an *index* to each component. Index terms are chosen among a predefined set of key terms, referred to as the *controlled vocabulary*. In some cases, we used automatic controlled-vocabulary indexing whereby a key word or phrase is assigned to an attribute if it occurs within the `text` field.

For a given vocabulary, the terms of the vocabulary (key words and phrases) may be organized along a conceptual hierarchy. Figure 3 shows excerpts of the conceptual hierarchies of key phrases for the attributes `ApplicationDomain` (Figure 3.a) and `Purpose` (Figure 3.b). Typically, for a given attribute, the keywords are organized in a single hierarchy whose root is the name of attribute itself. Notice that the `ApplicationDomain` hierarchy of key phrases is inspired from the (ACM) Computing Reviews 's classification structure [ACM85]. The hierarchical relationship between key phrases is a loose form of generalization, commonly referred to in information retrieval as "Broader-Term" [SM83].

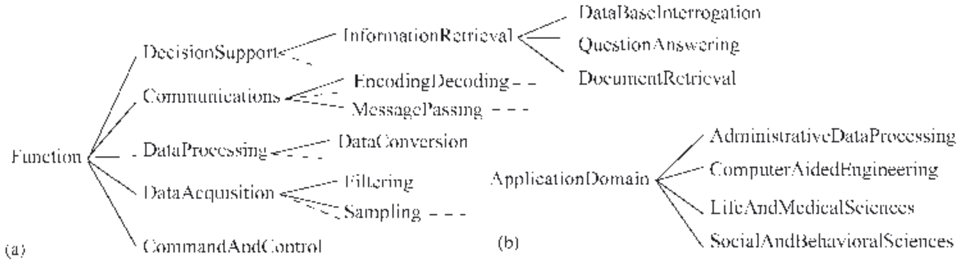


Fig. 3: Hierarchies of key phrases for the attributes `Purpose` and `ApplicationDomain`.

In contrast, the basic premise behind automatic indexing is that the words that occur in the document with a certain statistical profile are good content indicators [SM83]. Thus, a document is described by a (possibly weighted) set of words or lexical units.

3.2 Full-text versus controlled-vocabulary retrieval

Different indexing strategies support different retrieval techniques. With a controlled vocabulary, information seekers can only form queries using terms from the vocabulary. With full-text indexing no such limitations hold. Given a natural language query Q, the free-text retrieval algorithm returns the set of components S whose documentation includes any of the *significant*[1] words in Q.

In addition, the relevance of a document/component could be assessed in different ways, thus leading to specific retrieval methods like boolean or weighted vector methods. In our case, attribute values (key words and phrases) are used in boolean retrieval whereby component attribute values are matched against required attribute values (queries). The hierarchical relationships within an indexing vocabulary are used to extend the basic retrieval

[1]Language auxiliary constructs and domain specific words of common use are excluded. Furthermore, words of both the documents and the query are reduced to their stems [FBY94].

algorithms by adding different degrees of matching (instead of *true* or *false*) between two key words, e.g., depending on the length of the path separating them in the hierarchy.

In its simplest form a query is a list of *attribute query terms* (AQTs), which are ANDed. An AQT consists of an attribute and a list of ORed key phrases. Each AQT is assigned a weight and cut-off point, used for weighted boolean retrieval:

- `Query ::= AQT | AQT AND Query`
- `AQT ::= Attribute Weight CutOff LofKeyPhrases`
- `LofKeyPhrases ::= KeyPhrase | KeyPhrase OR LofKeyPhrases`

An AQT is a four-tuple $\langle Attribute, Weight, CutOff, LofKeyPhrases\rangle$ which retrieves the components C such that $Attribute(C) \cap LofKeyPhrases \neq \emptyset$. The query denoted by the tuple $(AQT_1, AQT_2, ..., AQT_k)$, returns the intersection of the sets of components retrieved by each individual AQT.

With weighted boolean retrieval, components are assigned numerical scores that measure the extent to which they satisfy the query, instead of being either "in" or "out". Let Q be a query with terms $(AQT_1, AQT_2, ..., AQT_k)$. The score of a component C is computed as follows:

$$Score(Q, C) = \frac{\sum_{i=1}^{k} Weight_i \times Score(AQT_i, C)}{\sum_{i=1}^{k} Weight_i} \quad (1)$$

where $Score(AQT_i, C)$ is 1.0 if $LofKeyPhrases \cap Attribute_i(C) \neq \emptyset$, and 0 otherwise.

3.3 A comparison

A number of studies have shown that free-text indexing and retrieval suffers from the differences in the terms the authors and searchers might use to refer to the same concepts [BM85]. In principle, this leads to lower recall (false negatives) and recall (false positives). With controlled vocabularies, both the indexer and the searcher will use the same vocabulary. However, controlled vocabulary indexing is labor intensive[2] and controlled vocabulary retrieval requires the searchers to familiarize themselves with the vocabulary. Hence, some authors have argued that difference in performances does not justify the cost [Sal86, FP94].

In *ClassServer*, we have implemented fully automatic free-text search and fully manual controlled vocabulary indexing and retrieval. Because the vocabulary is hierarchical, the purely Boolean retrieval could be extended to take into account the hierarchical relationships between terms of the vocabulary [Ost92]. We also provided a browsing tool that allows users to navigate through the vocabulary and find a concept of interest.

Because of the cost involved in controlled vocabulary indexing and retrieval, we adopted a two-layered approach. On the one hand, we provide searchers with tools to explore and navigate hierarchical vocabularies. On the other hand, we automate as much of the controlled vocabulary indexing as possible, which means we developed methods to help:

- identify the important concepts within a domain of discourse,
- organize them in a hierarchy, and
- index documents with the resulting vocabulary.

The last step is independent of the first two and may therefore be used with any vocabulary whether it is fully manually built or semi-automatic.

[2]Besides the cost of reading the documents one by one and manually assigning index terms to them, there is the cost of building and maintaining that vocabulary.

4 Automatic vocabulary constructing and indexing

The available documentation seemed to be the right place to search for important concepts: although there was a risk of getting a partial view of the underlying domain, and of depending too much on the terminology used by the documenter, it was clear that no meaningful concepts will be missed. The first step consists of determining the right lexical unit that corresponds to key concepts and then extracting all such units from the text.

4.1 Identifying the important concepts within a domain

Full text indexing uses the occurrences of words in document collections as indicators of the word's usefulness as content descriptor. The key problem with such an approach is that words are often taken out of context and are not constrained in any manner. Further, we argue that in computer science, as in any relatively new field, most of the important concepts are described by noun phrases, as in "Software Engineering" "Bubble Sort",etc.

Automatic approach

In order to extract those higher level lexical units, to which we, abusively, refer as *noun phrases*, we used *Xerox Part Of Speech Tagger* (XPost) [Cut92], a tool that tags the words of a natural language text by their respective syntactic ("part of speech") categories. For example, it assigns to "The common memory pool" the tag sequence `article adjective noun noun` (denoted by "at jj nn nn"). Analysis in XPost relies on: 1) a "tag table", mapping tokens to sets of tags, and 2) a probabilistic (Markovian) model of plausible tag sequences (induced at a preliminary step using unsupervised learning techniques). Thus, XPost is initially run on a training sample so that the tag sequences for interesting noun phrases could be identified. After describing the set of identified sequences by a (conservative) set of regular expression, XPost is run on the effective document corpus with its output piped to a tool for recognizing those expressions. The following table shows the regular expressions in an awk-like format:

```
PREFIX            (JJ | VBG   VBN)(NN | NNS | NP   NPS) | (NN | NNS   NP | NPS)
                  //Example "Small System", or "system", but not "small"
BASIC      ≡      PREFIX (JJ   NN | NNS | NP | NPS   VBG | VBN)*
                  //Example "Event based systems"
X_OF_Y     _      BASIC IN BASIC
                  //Example "Memory management of event based systems"
X_OF_A_Y   ≡      BASIC IN AT BASIC
                  //Example "storage requirements of an event based system"
```

In case a sentence matched several expressions, the longest running expression is taken. The resulting list may then be filtered based on frequency of occurrences in document collection.

Results

When applied to the *OTC* library, the extraction process identified 2,616 unique "noun phrases", with overall occurrences ranging from 163 (for the word "function") to 1, with 1,765 phrases occurring just once, including phrases such as "command line options" or "Conversion operator to a standard pointer". Phrases occurring too often are poor discriminators whereas those occurring rarely may not be important. Therefore, we discarded 8 "phrases" of occurrence higher than 100, e.g., OTC (267 times), "Function" (163) and

"String" (134), and a large set of phrases that occurred less than five times. For the experiment we used the remaining 229 phrases including some C++ identifiers that could have been removed manually. However, since the identification of such terms could not be easily automated, we left them in.

In a different experiment setting, we applied our algorithm to the *GenBank* (Genetic Sequence DataBank) [Bil86]. Overall, the experiments showed that while the hierarchy may not be "user-friendly" or make as much sense as a manually-built one, it can perform useful retrieval tasks equally well [MR87].

4.2 Constructing a hierarchy

A hierarchically ordered domain vocabulary may prove useful in various ways. For instance, it helps "librarians" locate the most appropriate term to describe a component, and "re-users" find the closest term to use in a query. Hierarchical relations may also be used to extend boolean retrieval methods to account for "close" matches (see [Ost92]). Constructing a thesaurus of domain concepts involves first identifying those concepts and then organizing them into a hierarchy.

Algorithm

Given a set of terms $T = \{t_1, t_2, ..., t_m\}$, a set of documents $D = \{d_1, d_2, ..., d_n\}$ with manually assigned indices $Idx() = \{t_{i_1}, t_{i_2}, ...\}$, we argued that [MR87]:

H_1 Terms that often co-occur in document indices are related,

H_2 The more frequently occurring a term, the more general its conceptual scope, and

H_3 If two terms co-occur often in document indices, whereby one of them has a more general scope than the other, than there is a good chance that the relationship between them is a generalization-like relationship.

The H_1 hypothesis is based on fact that documents tend to exhibit conceptual cohesion and logic, and because index terms reflect the important concepts within a document, they tend to be related. H_2 is based on observations made about terms occurring in both free-format natural language [Jon80] and indexes [WC85].

In [MR87], we suggested an algorithm based on H_1 to H_3 which, given a set of index terms $t_1, t_2, ..., t_m$ with their co-occurrence rates, generates a rooted acyclic graph.

```
1: Rank the index terms by decreasing order of frequency
2: Build the co-occurrence matrix $M$ ($i^{th}$ row = $i^{th}$ most frequent term)
3: Normalize the elements $M(i,j)$ (divide by $\sqrt{M(i,i) \times M(j,j)}$)
4: Choose terms $t_1$ through $t_l$ to include in the first level of the hierarchy
5: for $i$ from $l+1$ to $m$ do
6:    $v = \max(\{M(i,1), ..., M(i,i-1)\})$
7:    Create a link between $t_i$ and all $t_j$ such that $M(i,j) = v$.
```

Algorithm 1: Building a hierarchy out of a set of concepts.

The choice of the first level nodes is quite arbitrary although, ultimately, it has a very little impact on the overall hierarchy.

Results

We used subsections in files (an average of 10 subsections per file) as documents, and for each document, instead of counting the co-occurrence of two phrases as 1, as would be expected with index terms, we take the minimum of the occurrences of the two phrases as their frequency of co-occurrence within that document.

The first run of the algorithm generated a hierarchy with 291 relations between 291 phrases, including the dummy root node. A couple of excerpts from the hierarchy are shown below

```
...
0.2.1.1.2.1 LENGTH
0.2.1.1.2.1.1 LENGTH OF THE STRING
0.2.1.1.2.1.2 CAPACITY
0.2.1.1.2.1.2.1 CAPACITY OF THE STRING
0.2.1.1.2.1.3 RANGE
...
0.2.1.1.2.3.2 B
0.2.1.1.2.3.2.1 CONVERSION
0.2.1.1.2.3.2.1.1 SOBJECT
0.2.1.1.2.3.2.1.2 CONVERSION OPERATOR
```

Notice the "term" `B`, which is a C++ identifier. It was tagged as noun because it occurred in the text where a subject/object was expected and with a frequency high enough to make it into the vocabulary. As mentioned earlier, such terms were left in the vocabulary to get an idea about what an unfiltered hierarchy would look like. The relationship between `CONVERSION` and `SOBJECT` is an interesting one. `SOBJECT` is the name of the class representing strings which supports several conversion operations. Although general conversion wouldn't necessarily associate with strings, in the context of this library, the association is important and useful.

Since no reference hierarchy was available, the result could be evaluated only qualitatively. To this purpose, the result hierarchy was presented to six subjects who were asked to mark, for each node, whether the node represented a valid concept from the domain of discourse, and in case it did, to label the node's relationship to its parent as one of: a) *has broader-term*, b) *related*, to indicate any other relationship, and c) *unrelated* (used when there was no apparent relationship). For instance, the relationship between `LENGTH OF THE STRING` and `LENGTH` in the above schema is *has-broader-term*, whereas `RANGE` and `LENGTH` are only *related*. The evaluation of the six subjects are summarized in Table 1 below. The second line shows the results obtained by re-deriving the hierarchy after the removal of all the invalid terms (26 of them, including C++ identifiers).

Hierarchy	**invalid terms**	**has-broader-term**	**related**	**unrelated**
With invalid terms	9	20	37	34
Without invalid terms	0	27	39	34
Links removed	26	8	19	28
Links added	0	17	13	18

Table 1: Evaluating the individual links created by the statistical algorithm (all figures in %).

These results are disappointing compared to the *GenBank* experiment [MR87], even after we remove manually the invalid terms from the input. There are many reasons for this, which are explained elsewhere. We performed an additional number of tests to refine the hierarchy or explain the quality of the results, but they proved inconclusive. For the purposes of the retrieval experiment (see the next section), the automatically generated hierarchy was used as a flat set of terms, and even then, it didn't prove useful.

4.3 Automatic indexing with controlled vocabulary

Traditionally, indexing documents with a controlled vocabulary is carried out manually.

Algorithm

We attempted to automate it, lest we lose *some* (but not *all*) of the advantages of controlled-vocabulary indexing. Thus, a document d was assigned a term $t = w_1 w_2 ... w_n$ if it contains (most of) its component words, consecutively ("$...w_1 w_2 ... w_n ...$"), or in close proximity ("$...w_1 n_1 n_2 w_2 w_3 ... w_n ...$"). In our implementation, we reduced the words of both the terms of the vocabulary and the words of the documents to their word stem by removing suffixes and word endings. Also, we used two tunable parameters for indexing: 1) *proximity*, and 2) *partial match threshold*, i.e., the ratio of the number of words found in a document to the total number of words of a term. The proximity parameter indicates how many words apart should words appear to be considered as parts of the same noun phrase (term).

Results

In order to separate the vocabulary control from the performance of automatic indexing *per se*, we indexed the in-line documentation of classes with the `ApplicationDomain` vocabulary, which was built manually. In particular, we used a threshold of 2/3 and a proximity of 5, i.e. a term was assigned to a document when at least 2/3 of the words of the term occurred in the textual part of the attribute, with no two words more than 5 words apart. We wanted to get an idea about "how often" terminology issues miss some important term assignments, and about the appropriateness of the indexing parameters (threshold and maximum word distance). Our evaluation takes into account what is in the vocabulary, and what is in the text, and the question was, given the same limited vocabulary and limited textual description, would a human being have done it any differently. We studied 80 textual descriptions varying in size from a single sentence such as "Do not define an implementation for this", to half a page of text. The results are summarized in the table below.

	exact	related	extraneous terms	missing (different)	missing words
Number of terms	42	3	6	11	24
% among assigned	82	6	12		
Coverage	52	4		14	30

Here the extraneous terms are terms that shouldn't have been assigned (false-positive). Examples include the indexer mistaking the verb "[this method] sets" for the word "Sets" (as in collections). Other examples of extraneous terms include a case where the indexer assigned the term "Copying Strings" to the sentence "This class does not make copies of the character strings it is given...". The *missing terms* are terms that a human indexer would have assigned, and fall into two categories: a) a *synonym* for the actual word(s) was used instead of the actual words, or b) the concept does not appear *verbatim*, but is *implicit*.

In summary, only 6% of the assigned terms were wrong, which should only minimally affect retrieval precision. The indexer seems to have missed a significant number of terms, 44%, but minor refinements could have reduced this figure. Besides, the effect of these "false-negative" term assignments on retrieval recall would be hard to estimate, as the retrievability of a component depends on the whole set of index terms.

5 Retrieval experiments

The documentation of the *OSE* library consisted of 13 HTML files, one of which giving an overview of the library, and the remaining 12 describing specific subsets of the library. The 13 files contained a total of 37,777 words (244 Kbytes). The experimental data set consisted of about 200 classes and 2000 methods from the *OSE* library, used in 11 different queries.

Both full-text and controlled vocabulary indexing and retrieval were examined. Components were classified with two attributes: `ApplicationDomain` and `Description`. `ApplicationDomain` was indexed manually with a manually-built vocabulary, whereas `Description`, was indexed automatically with the automatically generated vocabulary (see Section 4.1) considered as a flat set (for reasons reported in Section 4.2).

Seven subjects, all of them experienced C++ programmers, participated in the experiment, although only the data from 5 subjects was usable. The subjects were given a questionnaire which included the statements of the queries, and blank spaces to enter the answer as a list of component names, much like an exam book. For each of the initial 77 (subject,query) pairs, we randomly assigned a search method (keyword-based versus plain text). For each (subject,query,search method) triplet, the subject could issue as many search statements as s/he wishes using the designated search, with no limitation on the time or on the number of search statements.

Query	**Full-text retrieval**			**Keyword retrieval**		
	Subjects	**% Recall**	**% Precision**	**Subjects**	**% Recall**	**% Precision**
1	3	100	88.666	2	50	50
2	4	50	100	1	50	100
3	1	100	100	4	100	100
4	1	100	80	4	50	100
5	4	25	12.5	1	0	0
6	3	33.333	33.333	2	12.5	25
7	2	65	75	3	66.333	50
8	2	30	75	3	30	83.333
9	3	53.333	100	2	30	78
10	3	78.333	80.333	1	35	100
Average	**(26)**	**63.49**	74.47	**(23)**	42.41	68.33

Table 2: Summary of retrieval results.

Table 2 shows recall and precision for the 11 queries. Initially, with the initial 7 subjects, for each query, we selected 3 subjects at random to perform the query using full-text retrieval, and 4 subjects to perform keyword retrieval, or *vice versa*. Later on, the results of two subjects proved globally useless and one query, the 11th, was dropped off since the three keyword-based answers were all rejected for various reasons.

Intuitively, it appears that plain-text retrieval yielded significantly better recall and somewhat better precision for the 10 queries (with a few exceptions). In order to validate these two results statistically, we performed a number of ANOVA tests, to check whether recall and precision were random variables, but the random hypothesis was rejected in both cases.

In summary, our experiments showed that: 1) those aspects of the pre-processing involved in controlled vocabulary methods that we automated were of poor enough quality that they were not used (the `Description` attribute), and 2) the fully automatic free text search performed better than the fully manual controlled-vocabulary based indexing and retrieval of components.

6 Discussion

Because the above results are somewhat counter-intuitive, we continue to analyze them, and to think of ways of improving the pre-processing involved in the controlled vocabulary-based methods. We hypothesize that multi-faceted classification and retrieval of reusable components to be at the wrong level of formality for the typical workflow of developers using a library of reusable components. We identify two very distinct search stages. The first stage is fairly exploratory, as developers do not yet know which form the solution to their problem will take, and a free-format search technique such as plain-text search is appropriate. Multi-faceted search may be too rigid and constraining for this early search step. This is even more so, considering that one might be searching components in several sites, each with its own representation conventions. The second search stage aims at selecting, among an initial set of potentially useful components, ones that will effectively solve the problem at hand. At this second stage, we need a far more detailed description of components and their inter-relationships than that provided by multi-faceted classification. To the extent that multi-faceted component retrieval retrieves a different set of components from plain text retrieval [FP94], the issue shouldn't be using one *or* the other, but using them *both*. Our work on reducing the pre-processing costs of multi-faceted component retrieval is a step in the right direction. We intend to explore ways of reducing the cost of formulating multi-faceted queries and of ensuring that they are used effectively. One such strategy consists of using case-based reasoning to generate queries based on those components returned by plain-text search that developers deemed relevant (on-going project).

References

[ACM85] ACM. An introduction to the cr classification system. *ACM Computing Reviews*, pages 45–57, January 1985.

[Bil86] A. H. Bilofsky, C. Burks, J. W. Fickett, W. B. Goad, F. I. Lewitter, W. P. Rindone, C. D. Swindell, and C. Tung. The GenBank Genetic Sequence Databank. *Nucleic Acids Research*, 14:1–4, 1986.

[BM85] D. Blair and M. E. Maron. An evaluation of y. *Communications of the ACM*, 28(3):289 299, 1985.

[Cut92] D. Cutting, J. Kupiec, J. Pedersen, and P. Sibun. A Practical Part-of-Speech Tagger. In *Proceedings of the Applied Natural Language Processing Conference*, 1992.

[Dum94] G. Dumpleton. *OSE - C + Library User Guide*. Dumpleton Software Consulting Pty Ltd., Parramatta, 2124, New South Wales, Australia, 1994.

[FBY94] W. B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1994.

[FP94] W. B. Frakes and T. Pole. An empirical study of representation methods for reusable software components. *IEEE Transactions on Software Engineering*, pages 1–23, August 1994.

[Hal93] R. J. Hall. Generalized behavior-based retrieval. In *Proceedings of the Fifteenth International Conference on Software Engineering. Baltimore, Maryland*, pages 371–380, May 1993.

[Jon80] K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. In B. C. Griffith, editor, *Key Papers in Information Science*, pages 305–315. Knowledge Industry Publications, Inc, White Plains (NY), 1980.

[MMM94] A. Mili, R. Mili, and R. Mittermeir. Storing and retrieving software components: A refinement-based approach. In *Proceedings of the Sixteenth International Conference on Software Engineering, Sorrento, Italy*, May 1994.

[MMM95] H. Mili, F. Mili, and A. Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, 1995.

[MR87] H. Mili and R. Rada. Building a Knowledge Base for Information Retrieval. In *Proceedings of the Third Annual Expert Systems in Government Conference*, pages 12–18. IEEE Press, October 1987.

[Mil94] H. Mili, R. Rada, W. Wang, K. Strickland, C. Boldyreff, L. Olsen, J. Witt, J. Heger, W. Scherr, and P. Elzer. Practitioner and SoftClass: A Comparative Study. *Journal of Systems and Software*, 27, May 1994.

[Ost92] E. Ostertag, J. Hendler, R. Prieto-Diaz, and C. Braun. Computing similarity in a reuse library system: An ai-based approach. *ACM Transactions on Software Engineering and Methodology*, 1(3):205–228, 1992.

[PD90] R. Prieto-Diaz. Integrating domain analysis and reuse in the software development process. In *Proceedings of the Third Annual Workshop on Methods and Tools for Reuse*, Syracuse (NY), June 1990. CASE Center, Syracuse University.

[PDF87] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, pages 6–16, January 1987.

[Sal86] G. Salton. Another Look at Automatic Text-Retrieval Systems. *Communications of the ACM*, 29(7):648–656, 1986.

[SM83] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.

[WC85] B. H. Weinberg and J. A. Cunningham. The Relationship between Term Specificity in MeSH and Online Postings in MEDLINE. *Bulletin Medical Library Association*, 73(4):365–372, 1985.

[ZW93] A. M. Zaremski and J. M. Wing. Signature matching: A key to reuse. *Software Engineering Notes*, 18(5):182–190, 1993.

[ZW95] A. M. Zaremski and J. M. Wing. Specification matching: A key to reuse. *Software Engineering Notes*, 21(5), 1995.