ELSEVIER

Information and Software Technology xx (0000) 1–17

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

An experiment in software component retrieval

Hafedh Mili*, Estelle Ah-Ki, Robert Godin, Hamid Mcheick

Département d'Informatique, Université du Québec à Montréal, Case Postale 8888 (A), Montréal, Que., PQ, Canada H3C 3P8

Received 29 September 2002; revised 11 November 2002; accepted 30 December 2002

Abstract

Our research centers around exploring methodologies for developing reusable software, and developing methods and tools for building inter-enterprise information systems with reusable components. In this paper, we focus on an experiment in which different component indexing and retrieval methods were tested. The results are surprising. Earlier work had often shown that controlled vocabulary indexing and retrieval performed better than full-text indexing and retrieval [IEEE Trans. Software Engng (1994) 1, IEEE Trans. Software Engng 17 (1991) 800], but the differences in performance were often so small that some questioned whether those differences were worth the much greater cost of controlled vocabulary indexing and retrieval [Commun. Assoc. Comput. Mach. 28 (1985) 289, Commun. Assoc. Comput. Mach. 29 (1986) 648]. In our experiment, we found that full-text indexing and retrieval of software components provided comparable precision but much better recall than controlled vocabulary indexing and retrieval of components. There are a number of explanations for this somewhat counter-intuitive result, including the nature of software artifacts, and the notion of relevance that was used in our experiment. We bring to the fore some fundamental questions related to reuse repositories.

Keywords: Software reuse; Multi-faceted classification; Boolean retrieval; Plain-text retrieval; Retrieval evaluation; Approximate retrieval

1. Introduction

1.1. Component retrieval: do we still care?

Software reuse is seen by many as an important factor in improving software development productivity and software products quality [2,13]. It is customary in the software reuse literature to make the distinction between the *generative* approach whereby developers reuse development *processors* such as code generators or high-level specification language interpreters, and the *building blocks approach*, whereby developers reuse the *product* of previous software development efforts in the process of building new ones. The building blocks approach modifies the traditional, analytical, divide and conquer approach to system specification and design by introducing three reuse tasks that must be performed before one falls back on analytical methods: (1) searching and retrieving reusable components based on partial specifications, (2) assessing the reuse worth of the retrieved components, and, possibly, (3) tailoring the reusable components to the specifics of the problem at hand [22]. In this paper, we focus on computer support for software component search and retrieval.

The problem of component retrieval has been widely addressed in the software reuse literature. A number of developments have rendered this problem somewhat uninteresting. From a technical point of view, research in the area has hit the formal methods cost barrier: the investment needed to get the next level of performance—to get beyond signature matching or multi-faceted classification—overshadowed the anticipated productivity gains. Second, there was a widespread recognition in the object-oriented reuse community that classes are too small units of reuse, for two reasons. First, classes cannot be reused in isolation. Second, considering that more is gained by reusing designs than by reusing code, individual classes embody mostly code, but little design. Finally, empirical evidence from reuse repositories had shown that small components may account for a good fraction of reuse instances, but in the end, account for little reuse volume,[1] and thus little benefit [12]. The underlying lesson was 'focus

* Corresponding author. Tel.: +1-514-987-3943; fax: +1-514-987-8477.
E-mail address: hafedh.mili@uqam.ca (H. Mili).

[1] Isoda reported on an experimental reuse program at NTT where they found that components of 50 lines or less accounted for 48% of the reuse instances and 6% of the reuse volume, while modules 1000 lines or larger accounted for only 6% of the reuse instances, but of 56% of the reuse volume [12].

0950-5849/03/$ - see front matter
doi:10.1016/S0950-5849(03)00002-8

on a small number of large components embodying design as well as code', i.e. application frameworks.

Interestingly, the Internet has brought repository issues back to the forefront. First, it has enabled a virtual market for software components: developers have been searching the web for software components, both free and for-fee, for the past decade. Second, inter-enterprise (B2B) electronic commerce relies on enterprises ability to 'plug-in' each other's systems to be able to complete transactions, end to end. The ability to plug systems together has become a major factor in entering into business relationships [2], some times the overriding one [26]. The pluggability of information systems for the purposes of entering into electronic commerce starts with the lookup of industry-wide registries of APIs exported by potential partners. Standards are emerging to represent such APIs in a technology independent way (see e.g. ebXML [26]), but the issue of conceptual appropriateness remains whole. Notwithstanding things such as ebXML registries or software vendor-specific web sites, it seems that much reuse is taking place in the unstructured world of the world wide web, as opposed to a corporate managed reuse repository with dedicated personnel and strict quality control. This paper explores component classification and retrieval methods with an overriding concern for automation.

1.2. The component retrieval problem

A wide range of component categorization and searching methods have been proposed in the literature, from the simple string search (see e.g. Ref. [21]), to faceted classification and retrieval (e.g. Refs. [27,28]) to signature matching (see e.g. Ref. [37]) to behavioral matching (see e.g. Refs. [10,17,38]). Different methods rely on more or less complex descriptions for both software components and search queries, and strike different trade-offs between performance and cost of implementation [22]; the cost of implementation involves both initial set-up costs, and the cost associated with formulating, executing and refining queries. In the context of our research, we developed four classes of retrieval algorithms (1) retrieval using full-text search on software documents and program files, (2) multi-faceted classification and retrieval of components, (3) navigation through the structure of components, and (4) signature matching. The first two use the documentation or the *meta-data* that accompanies software components, and thus rely on its existence, its quality, and some pre-processing. The last two focus on the *structure* of the software components themselves, and thus depend on the availability of that structure in some form—source code, interface—and the availability of (computer) language processors.

An age-old debate, first in the information retrieval literature [4,31], and later in the context of reuse repositories [6,8,16,23], has opposed the free-text classification and retrieval of components to the so-called controlled vocabulary, multi-faceted classification and retrieval of components. The conventional wisdom is that free-text retrieval costs nothing—no manual labour—but produces many false positives (matches words taken out of context) and false negatives (misses out relevant components because of the use of a non-standard terminology). Controlled-vocabulary indexing and retrieval is supposed to solve both problems by providing a common vocabulary for classification and retrieval, and by having actual human beings classify documents/components. However, it involves a major cost in building and maintaining such vocabularies and in classifying/indexing components. Research in the area has traditionally attempted to bridge the gap between the two approaches in terms of cost and performance. From the free-text end, research has aimed at making the matching more intelligent and less dependent on surface-level similarity, but keeping humans out of the loop—e.g. using associations between terms instead of term matching or identity, as in latent semantic analysis methods [6,11,16]. From the controlled vocabulary end, research has aimed at automating or assisting the manual steps, but hopefully without losing much in terms of quality of retrieval. Our own work has covered both approaches, and this paper reports on a number of experiments trying out different ideas and comparing approaches.

Our first experiment dealt with the construction of domain vocabularies. Much of the earlier work on automated indexing of textual documents had relied on the statistics of the occurrences (and co-occurrences) of key terms or phrases within document collections to infer content indicators for documents and relations between key terms [15,30]. Our work furthers these ideas to build concept hierarchies based on statistics of (co)occurrences alone. A technique that worked well in previous experiments was less successful with software documentation. The experiment is described, and the results are analyzed in Section 3. The second experiment dealt with the automatic indexing of software components (their documentation) using a controlled vocabulary: the basic idea is that an index term (say 'Database Management Systems') is assigned to a component if 'most' of its constituent words appear 'close' to each other within the documentation of the component; most and close are both tunable parameters of the method. In principle, the automatic assignment of index terms suffers from the same problems as free text search: matching words out of context (false positives), and missing out on relevant components because of choice of terminology (false negatives). However, we felt that the use of compound terms would reduce the chances of false positives, and the use of inexact matches (most, close) would reduce the chances of false negatives. The results bear this out, and are discussed in Section 4.

Our third experiment consisted of comparing an all-manual controlled vocabulary indexing and retrieval method with an all-automatic free-text indexing and retrieval method, using a variant of the traditional information retrieval measures, *recall* and *precision*. Instead of computing recall and precision based on some abstract measure of 'relevance', as is done in information retrieval and in most reuse library experiments, we adapted the measure to take into account the true utility of the retrieved components to solve the problem at hand. Further, we used a realistic experimental protocol, one that is closer to the way such tools would be used in practice. Here the results were surprising. Full-text retrieval yielded *significantly* better recall and somewhat better precision—although the difference is statistically insignificant. The experiment is described in Section 5. We analyze the results in light of new evidence about the behavior of users in an information retrieval setting. We conjecture that multi-faceted retrieval requires more information than the user is able to provide in the early stages of problem solving, and fails to capture a faithful expression of users' needs at the later stages.

Section 2 provides a brief introduction to our tool set. We conclude in Section 6.

2. ClassServer: an experimental component repository

2.1. Overview

This work is part of ongoing research at the University of Québec at Montréal aiming at developing methods and tools for developing reusable software, and for developing with reusable software. The work described in this paper centers around a tool kit called *ClassServer* that consists of various tools for classifying, retrieving, navigating, and presenting reusable components (see Fig. 1). Reusable 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 supporting the various reuse tasks. So far, we have developed extractors for Smalltalk and C++. The information extracted by these tools consists of built-in language structures, such as *classes*, *variables*, *functions*, and *function parameters*. To these, we added a representation for *object frameworks*, which are class-like object aggregates that are used to represent application frameworks and design patterns [24]; unlike the built-in language structures, which are extracted by parsers, object frameworks need to be manually encoded. Fig. 1 shows a very schematic view of the ClassServer tool set. The tool set may be seen as consisting of three subsystems. The first subsystem, labeled 'Full-text retrieval' supports the required functionalities for full-text retrieval of source code *files*, namely, the 'Full-text indexer', and the 'Full-text search tool'. Their functionalities are explained in Section 2.3.1.

The component browser and the keyword retrieval subsystems use the structured representation of the components that is extracted by the tool referred to as 'semantic/structural parser' in Fig. 1. Typically, the parsing produces a trace of the traversal of the abstract syntax tree. The trace consists of a batch of component creation commands (in Smalltalk), which are executed when we 'load' the trace; that is the *structured component loader*. Each kind of component is defined by a descriptive template that includes: (1) structural information describing the kind of subcomponents a component can or must have (e.g. a *class* has *variables* and *methods*, a *framework* has *participants*, *message sequences*, etc.), (2) code, which is a string containing the definition or declaration of the component in the implementing language, and (3) descriptive attributes, which are used for search purposes; for

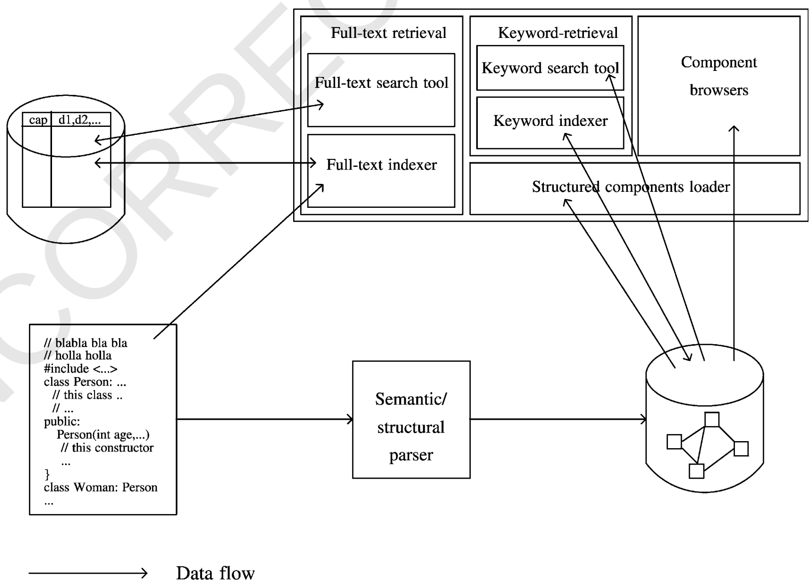

Fig. 1. Overall architecture of ClassServer.

example, a *class* has an *author* and an *application domain*, a *method* has a *purpose*, etc. Descriptive attributes, or simply, attributes, represent non-structural, non-intrinsic properties of software components, and are often derived from non-code information such as documentation, or entered explicitly by the person(s) responsible for managing the component library. Attributes will be described in more detail in Section 2.2.

2.2. A multi-faceted classification of components

Attributes are used in ClassServer to represent categorization/classification facets, as in Prieto-Diaz's multi-faceted categorization of components [28]. Attributes are themselves objects with two properties of their own: (1) *text*, which is a (natural language) textual description, and (2) *values*, which is a collection of key words or phrases, taken from a predefined set referred to as the *vocabulary of the attribute*. The text is used mainly for human consumption and for documentation generation [21]. Filling in the *values* property is referred to as *classification*, *categorization* or *indexing*. When human experts assign those key words or phrases from a predefined list, we talk about *manual controlled-vocabulary indexing* [30]. In our case, we used *automatic controlled-vocabulary indexing* whereby a key word or phrase is assigned to an attribute if it occurs within the text field. More on this in Section 4.

For a given attribute multiple values are considered to be *alternative* values (ORed), rather than *partial* values (ANDed). For example, for the attribute 'Purpose' of a component, several values mean that the component has *many* purposes, and *not* a single purpose defined by the conjunction of several terms. For a given vocabulary, the terms of the vocabulary (key words and phrases) may be organized along a conceptual hierarchy. Fig. 2 shows excerpts of the conceptual hierarchies of key phrases for the attributes 'Application Domain' (Fig. 2a) and Purpose (Fig. 2b). Notice that the Application Domain hierarchy of key phrases is inspired from the (ACM) *Computing Reviews*'s classification structure [1]. The hierarchical relationship between key phrases is a loose form of generalization, commonly referred to in information retrieval as 'Broader-Term' [30]. Attribute values (key words and phrases) are used in boolean retrieval whereby component attribute values are matched against required attribute values (queries, see below). The hierarchical relationships within an indexing vocabulary are used to extend the basic retrieval algorithms, as explained in Section 2.3.2.

2.3. Software component retrieval in ClassServer

As mentioned earlier, ClassServer provides two methods of classifying (and retrieving) software components, namely, free-text indexing and search of software components (source code and documentation), and multi-faceted classification and retrieval of components. We describe them both briefly below.

2.3.1. Free text indexing and search

By free-text indexing, we refer to the class of methods whereby the contents of a document are described by a weighted set of words or lexical units occurring in the document. Different methods use different selection mechanisms to restrict the set of eligible content indicators, and different weighting schemes [30]; the algorithm we used does not use a weighting scheme. Let us assume for the moment that *all* the words found in a document are used as potential content indicators. Given a natural language query Q, the free-text retrieval algorithm returns the set of components S computed as follows:

(0) Break the query Q into its component words $w_1, \ldots, w_n$,
(1) $S \leftarrow$ set of components whose documentation included w_1,
(2) **For** $i = 2$ **To** n **Do**
(2.1) $S_i \leftarrow$ set of components whose documentation included w_i,
(2.2) $S \leftarrow S \cap S_i$

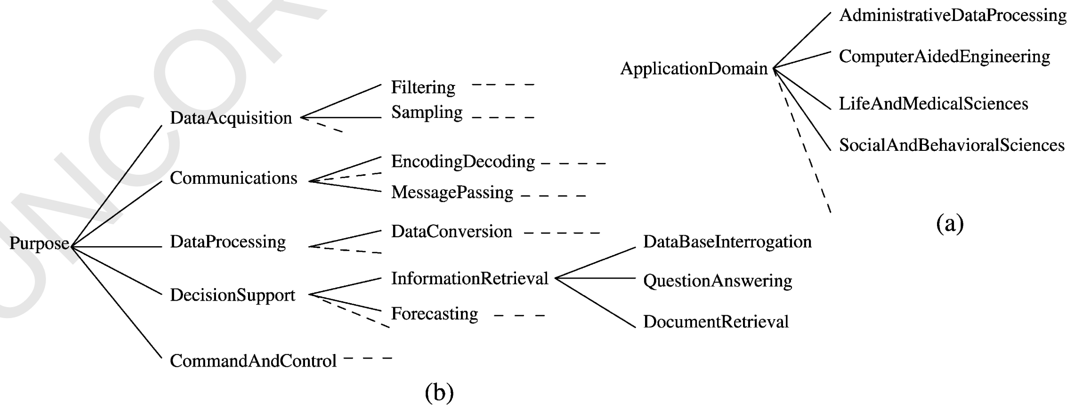

Fig. 2. Hierarchies of key phrases for the attributes Application Domain and Purpose.

When we break a query into its component words, we exclude all the words that are not significant in the application domain. This includes common language words such as 'the', 'an', 'before', and so forth. It also includes domain specific words that are likely to be found in every document—software component documentation in this case. For example, we would expect the word 'computer' to appear everywhere in a computer science collection. These are called *stop* words. In order to account for lexical variations when matching words of the query to words of the documents, we reduce both to their roots, as in mapping 'Managing' and 'Management' to 'Manag'. Algorithms to perform this mapping are called *word stemmers*, and we used one published in Ref. [9]. Finally, to speed search, we pre-process the entire document collection by creating an inverted list which is a table whose keys are unique words stems such as Manag, and whose values are lists of the documents in which the word occurred in one lexical form or another (e.g. as Managing or Management). This reduces the step (2.1) above to a simple table look-up.

2.3.2. Multi-faceted controlled-vocabulary retrieval

Our choice for the representation of queries involved a trade-off between flexibility and expressiveness, on the one hand, and allowing users to specify the most common queries most easily and most efficiently, on the other. The simplest form of a query is a list of so called *attribute query terms* (AQTs), considered to be ANDed. In its simplest form, an AQT consists of an attribute, and a list of key phrases, considered to be ORed. In the actual implementation, each AQT is assigned a weight and cut-off point, used for weighted boolean retrieval and conceptual distance, respectively (see below). Symbolically:

* Query :: = AQT|AQT **AND** Query
* AQT :: = Attribute Weight CutOff ListOfKeyPhrases
* ListOfKeyPhrases :: = KeyPhrase|KeyPhrase **OR** ListOfKeyPhrases

A single AQT retrieves the components whose attribute ⟨Attribute⟩ has at least one value in common with ⟨ListOfKeyPhrases⟩. Viewing attributes as functions, an AQT denoted by the four-tuple ⟨Attribute, Weight, Cut Off, ListOfKeyPhrases⟩ retrieves the components C such that Attribute(C) ∩ ListOfKeyPhrases ≠ Φ. The query denoted by the tuple ($AQT_1,\ldots,AQT_k$), returns the intersection of sets of components that would have been returned by the individual AQTs.

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 ($AQT_1,\ldots,AQT_k$), where $AQT_i = \langle \text{Attribute}_i, \text{Weight}_i, \text{CutOff}_i, \text{ListOfKeyPhrases}_i \rangle$. The score of a component C is computed as follows:

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

where Score(AQT_i,C) equals 1.0 if $\text{ListOfKeyPhrases}_i \cap \text{Attribute}_i(C) \neq \Phi$, and 0 otherwise.

Another extension meant to handle approximate matches is based on the number of edges separating the key terms of the query from the key terms of the attribute of the component in the conceptual hierarchies that enclose them (as in Fig. 2). If, for some i, $\text{ListOfKeyPhrases}_i \cap \text{Attribute}_i(C) \neq \Phi$, we look at some aggregate of the path lengths that separate elements of $\text{ListOfKeyPhrases}_i$ from elements of $\text{Attribute}_i(C)$ and use that to assign a score between 0 and 1 for the query term; the higher the average distance, the lower the score. The mathematical properties of the resulting similarity metric—called DISTANCE—and its effectiveness at emulating human relevance judgements have been thoroughly documented in Ref. [29]. In ClassServer, the cut-off value puts an upper limit on the path lengths to be considered in the computation; key phrases that are separated by more than 'cut-off' edges are considered totally unrelated.[2] A third extension uses the hierarchical relationships between key terms to 'classify' the query within a virtual classification structure of components that is based on the relationships between their attribute values, returning the most 'specific' components that are more 'general' than the query. The 'specialization' relationship has a formal meaning in this case [17]. Neither of the last two extensions was used in the experiments of Section 5, and will not be discussed further.

2.4. The component library

For the purposes of the experiment described in Section 5, we loaded the ClassServer repository with the OSE library [7] which contained 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 (see Section 2.3.1). Further, a shell script put the files through a C++ pre-processor before they were input into the C++ extractor (see Section 2.1). Because of the good quality and format consistency of the in-line documentation (comparable to Javadoc), we were able to automatically assign C++ comments as text values for the 'Description' attribute of various components (classes, methods, variables). Overall, we classified components using two attributes ApplicationDomain, and Description. ApplicationDomain was indexed manually, but in a fairly systematic fashion, using the on-

[2] This 'sunsetting' is used to fix some singularities in the otherwise well-behaved similarity metric [29].

line documentation of the library. In fact, the section headers of the documentation were themselves used as index terms (see Ref. [20] for a justification). The documentation grouped the various classes by application area. Further, each class was first described by a general statement about what the class does, followed by a more detailed description of its services, which mapped closely to methods. Some utility methods were not documented, and we could not assign those an ApplicationDomain; however, all classes were properly classified.

For the Description attribute (telling what a component does and how it does it, rather than 'what it is used for'), we did not have a ready-made indexing vocabulary. We considered using available classification structures that include computer science concepts, including the 1200 + terms Computing Reviews classification structure [1]. However, the classification terms were too general to be of any use to our library of components. For example, whereas we needed terms that corresponded to the different sorting algorithms ('MergeSort', 'RadixSort'), the term 'Sorting' was a leaf node of the ACM hierarchy. Accordingly, we decided to develop our own vocabulary by analyzing the available software documentation; the process of building the vocabulary is described next. Further, we decided to perform the actual indexing of the attribute (the assignment of key terms to attribute values) automatically. The algorithm and the results are discussed in Section 4.

3. Constructing domain vocabulary

A hierarchy of the important concepts in a domain has many uses in the context of software component retrieval. In addition to the advantages of having a standard vocabulary, its hierarchical structure helps 'librarians' locate the most appropriate term to describe a component, and 're-users' find the closest term to their need to use in a search query. Those same relations may also be used to extend boolean retrieval methods to account for 'close' matches, as shown in Section 2.3.2 (see also Refs. [11,27]). Constructing a hierarchy of the important concepts in a domain (or thesaurus) involves identifying those important concepts and their preferred terminology (Section 3.1), and organizing them into a hierarchy (Section 3.2). We discuss these issues in turn.

3.1. Extracting a set of concepts

A good place to look for the important computer science concepts that are germane to a library of reusable components is the documentation of the library itself. By looking *only* at the documentation, we run the risk of getting a partial and narrow view of the underlying domain, and of depending too much on the terminology used by the documenter. At the same time, we are assured that we will not miss any concepts that are important to the particular library (or libraries) at hand.

The next question is one of identifying the right lexical unit that corresponds to key concepts, and extracting such units from the text. Computer science being a relatively new field, most of the important concepts are described by noun phrases, as in 'Software Engineering' 'Bubble Sort', 'Printing Monitor', and so forth, rather than single words as is the case for more mature fields such as medicine.[3] In order to extract those higher level lexical units, to which we will abusively refer as noun phrases, we used *Xerox Part Of Speech Tagger* (XPost) [5]. XPost is a program that takes as input a natural language text and produces the syntactic ('part of speech') category or *tag* for each word or *token* for the text. For example, it assigns to the phrase 'The common memory pool' the tag sequence 'at jj nn nn', where 'at' stands for *article*, 'jj' for *adjective*, and 'nn' for *noun*. XPost uses two major sources of information to assign tags to words of a sentence: (1) a 'tag table', giving the set of tags that correspond to a given token, and (2) a probabilistic (markovian) model of the allowable sequences of tags. For example, the word 'book' can be a noun ('nn') or a verb ('vb'). If we also know that 'the' is an article and that only nouns can follow articles, we know that 'book' in the phrase 'the book' is a noun. XPost falls within the category of parts of speech taggers that derive the probabilistic model using unsupervised learning [5].

One of the typical uses of XPost is to extract phrases that follow a given pattern. We used XPost to extract 'noun phrases' that are likely to represent important domain concepts. To this end, we ran XPost on a training sample, we identified the tag sequences for the noun phrases in which we were interested, and then looked for a set of regular expressions that would have extracted those phrases. Those regular expression were then used to filter the output of XPost to extract noun phrases. The first set of regular expressions (a grammar) accepted far too many phrases, and we had to refine the grammar through trial and error, with a bias towards minimizing false positive phrases, at the expense of missing out some valid phrases. Fig. 3 shows the regular expressions in an awk-like format.

In case a sentence matched several expressions, we take the longest running expression. For example, if we analyze the sentence 'Memory management of event based systems', we produce a single noun phrase consisting of the entire sentence, rather than the two phrases 'Memory management' and 'event based systems', both of which matching the pattern BASIC.

We used this approach on the on-line documentation of the library. The documentation 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

[3] See Ref. [39] for a discussion on the evolution of languages and terminology.

```
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"
```

Fig. 3. Grammar for noun phrases.

extraction process identified 2616 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'. Typically, phrases that occur too often are not good discriminators [30]. Further, phrases that occur rarely may not be important for the domain at hand. We found 8 'phrases' that occurred more than a 100 times, and discarded them: OTC (the name of the library, 267 times), 'Function' (163), 'String' (134), 'Member function' (114), 'Object' (114), 'Class' (113), 'Example' (105), and 'Program' (104). We also discarded the phrases that occurred less than five times. Overall, we used 229 phrases. These include C++ identifiers that may have appeared in the code examples, and possibly referred to thereafter in the running text. We could have removed them from the vocabulary that was fed into the hierarchy builder, but we chose to exclude any manual processing or decisions that cannot be systematized or automated.

3.2. Constructing a hierarchy of important domain concepts

Having identified a set of the important concepts in a domain, we need to organize those concepts in a conceptual hierarchy. We present a simple algorithm that does just that based on statistics of occurrences of these concepts in documents. Next, we describe an earlier experiment with the algorithm that provided encouraging results. We conclude with the results of the algorithm on the set of concepts extracted with the method described in Section 3.1.

3.2.1. Principles

Given a set of terms $T = \{t_1, \ldots, t_m\}$, a set of documents $D = \{d_1, \ldots, d_m\}$ with manually assigned indices $\mathrm{Idx}(d_i) = \{t_{i_1}, t_{i_2}, \ldots\}$, we argued that [18]:

- H_1 Terms that co-occurred often in document indices were related in a way that is important to the domain of discourse,
- 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 (and thus are related, according to H_1), and if one has a more general scope than the other, than there is a good chance that the relationship between them is a generalization/specialization-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. The second hypothesis is based on observations made about both terms occurring in free-format natural language [14] as well as index terms [34].

We developed an algorithm that generates an acyclic graph with a single node with in-degree 0 (root) based on the above hypotheses [18]. Given m index terms $t_1, \ldots, t_m$, the algorithm operates as follows:

(1) Rank the index terms by decreasing order of frequency,
(2) Build a matrix of co-occurrences (call it M) where the ith row (column) corresponds to the ith most frequent term,
(3) Normalize the elements of the matrix M by dividing $M(i,j)$ by the square root of $M(i,i) \times M(j,j)$; note that after this normalization, $M(i,j) \leq 1$,
(4) Choose terms to include in the first level of the hierarchy; assume that the terms t_1 through t_{l_i} were chosen to be included in the first level,
(5) For $i = l_1 + 1$ through m
5.1 Find the maximum of the elements $M(i,1)$ through $M(i, i-1)$. Note that because of the ordering of rows and columns (step 2), these are the frequencies of co-occurrences of t_i with the terms whose occurrences are higher than that of t_i,
5.2 Create a link between the term t_i and all the terms t_j such that $j < i$ and $M(i,j) =$ maximum found in 5.1.

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

3.2.2. A case-study: the Genbank experiment

In one experiment, we used the *GenBank* genetic sequences database (databank). The GenBank Genetic Sequence DataBank serves as a repository for genetic sequences [3]. The entry for each sequence includes, among other things, the article that reported the discovery of the sequence, and a set of keywords that describe the sequence, including names of components or processes that are involved either in the composition and transformation of the sequence, or it is discovery. For our purposes, each entry corresponded to a document. We ran the experiment on 5700 such 'documents'. The co-occurrences matrix was limited to those keywords that occurred more than 10 times, and there were 274 of those. The resulting hierarchy was evaluated both qualitatively and quantitatively. The qualitative evaluation had to do with whether the parent-child links that were created were meaningful in general (hypothesis H_2), and whether they were generalization/specialization-like in particular (hypothesis H_3). Experts found that 50% of the links were indeed 'generalization/specialization' (G/S), as in the link between 'Heavy Chain Immunoglobulins' and 'Immunoglobulins'. Another 15% of the links were deemed meaningful as when the two terms represent a chemical component, and the process that creates it. The remaining 35% could not be characterized. Clearly, the resulting hierarchy was by far not as 'coherent' or 'enlightening' as manually built hierarchies such as the Computing Reviews Classification Structure, for example.

The quantitative evaluation had to do with the extent to which the resulting hierarchy supported extended boolean retrieval (DISTANCE-based, see Section 2.3.2) of documents any better or worse than a manually built hierarchy[4] that contained the same terms. For a given hierarchy H, the evaluation consists of: (1) using DISTANCE on H to rank a set of documents by order of relevance with respect to a set of queries, (2) asking human subjects to do the same, and (3) computing the correlation between the two rankings; the higher the correlation, the more faithful is distance to human evaluation, and the more useful is the hierarchy. Our experiments showed that the automatically constructed hierarchy performed as well, if not better than the manually built one [18].

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. We had observed that the keywords did not belong to a single conceptual domain, and that across-domain relationships could dominate within-domain ones. An algorithm that focuses on the strongest relationships would miss potential generalization relationships. For example, we had chemicals as well as chemical processes, and we had hypothesized (but not tested) that, had we separated them and applied the algorithm to the separate sets, we might have gotten more consistent hierarchies [18]. In other words, we felt that there was room for improvement.

[4] The Medical Subject Headings hierarchy, maintained by the National Library of Medicine, and used to support its on-line bibliographic retrieval system MEDLINE [32].

3.2.3. Constructing the graph based on OSEs on-line documentation

The construction of the hierarchy requires co-occurrence data between phrases within relatively coherent text units. We can break the documentation different ways, where a 'document' may be either, an entire file, a major section within a file, a subsection within a file, or even a paragraph. Whatever the document, we have to make sure that: (1) the phrases are good content indicators for that document, and (2) the co-occurrence of two phrases within the same document is not fortuitous and does reflect a significant relationship. The first constraint may suggest that we use documents that are big enough that phrase occurrence statistics become significant. The second constraint suggests that we use documents that are small enough that phrase co-occurrence be confined to a coherent textual unit. We decided to use subsections in files (an average of 10 subsections per file) as documents. Further, for each document, if a phrase P_1 occurred m times and a phrase P_2 occurred n, we consider that the phrases co-occurred $\mathrm{minimum}(m, n)$ times.

The first run of the algorithm generated a hierarchy with 291 relations between 291 phrases, including the dummy root node. Because we had no other hierarchy to which to compare it on a specific task, as was the case for the experiment described in Section 4.1, we could only evaluate the hierarchy *qualitatively*. To this end, we presented six subjects with the hierarchy and asked them 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* [33], which is a loose form of generalization, (b) *related*, to indicate any relationship other than has broader term, and (c) unrelated. *Unrelated* was used when there was no apparent relationship between a node and its parent. We show below excerpts from the hierarchy to illustrate the three kinds of relations. The relationship between LENGTH OF THE STRING and LENGTH is *has-broader-term*. That between RANGE and LENGTH is *related*.

...
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

We note the 'term' B, which is a C++ identifier that was tagged by XPost as a noun, because it is not a known verb or noun, and because it occurred in the text where a subject/object was expected. B occurred enough times to make it into the vocabulary. As mentioned earlier, we decided to leave such terms in to get an idea about what the hierarchy would look like without any manual filtering. In this case, not only B should not have been there, but all of the relationships between B and its children (CONVERSION) are non-significant. Such relationships are labeled as unrelated. The relationship between CONVERSION and SOBJECT is an interesting one. SOBJECT is the name of the class representing strings. This class supports several conversion operations, and hence the association. Somebody thinking of CONVERSION in general, would not think of strings. However, in the context of this library, the association is important and useful. This is similar to the kind of indirect associations between keywords exploited by the CODEFINDER system [11], which reflect the structure of the library as much as it reflects the structure of the semantic domain.

The evaluation of the six subjects are summarized in Table 1. The second line shows the results obtained by rederiving the hierarchy after we have removed the invalid terms (26 of them). Notice that because not all 26 terms were leaf nodes, by removing them we needed to reassign parents to 18 valid terms.

These results are disappointing compared to those obtained in the GenBank experiment [18], even after we remove manually the invalid terms from the input. The reasons are easy to identify. In the GenBank experiment the terms of the hierarchy did indeed describe important concepts in the domain, as opposed to the indiscriminate noun phrases extracted from our software documentation. We attempted a number of refinements using statistical measures to eliminate 'spurious' terms. Our first attempt was to eliminate the terms with the lowest frequency (5). This reduced the number of terms from 291 to 194, but ironically, only one non-applicable term was eliminated, and the distribution of the remaining relationships (has-broader-term, related, and unrelated) remained about the same. We used another measure of the *information value* carried by a given term, i.e. the extent to which it differentiates a specific and relatively small subgroup of the document set. Let T be a term, and d a document, we define $\mathrm{FREQ}(T,d)$ as the number of occurrences of T in d, and $\mathrm{FREQ}(T)$ as the total number of occurrences of T. The

Table 1
Evaluating the individual links created by the statistical algorithm

Hierarchy	Percentage of invalid terms	Percentage of has-broader-term	Percentage of related	Percentage of 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 2
Evaluating the hierarchy after filtering the terms that occurred more than 20 times, and whose entropy is more than half of the maximum possible entropy

Percentage of non-app. term	Percentage of has-broader-term	Percentage of related	Percentage of unrelated
7	19	36	38

entropy of a term T is defined as follows:

$$\mathrm{ENTROPY(T)} = \sum_{d \in \mathrm{Documents}} \frac{\mathrm{FREQ}(T,d)}{\mathrm{FREQ}(T)} \times \log \frac{\mathrm{FREQ}(T)}{\mathrm{FREQ}(T,d)}$$

For a given number of occurrences $\mathrm{FREQ}(T) = N$, the entropy is maximal if N are evenly spread across the document collection. If there are N documents, that entropy is $\log(N)$, and it correspond to T occurring exactly once in each of N documents. Let MAXENTROPY(T) be that maximum. Generally speaking, good terms are the ones with the smallest spread possible, i.e. whose entropy is closest to zero. Accordingly, we filtered the terms based on the ratio ENTROPY(T)/MAXENTROPY(T) : among the terms that occurred more than a threshold frequency[5] F_0, we rejected the ones for which the above ratio is above a certain threshold ρ. We tried several values of F_0, and several values of ρ. For $F_0 = 20$, and $\rho = 0.75$, 0.666, and 0.5, we eliminated 14, 22, and 37 terms, respectively, from the initial set of 194 terms. Table 2 shows an evaluation of the relationships within the generated hierarchy when $F_0 = 20$, and $\rho = 0.5$.

By looking at the remaining list of terms, a considerable number remain that should not be there. Hence, this test is not very effective at filtering invalid terms.

The second explanation for these results is related to the size of the document set. The GenBank experiment used 5700 documents, while this one used 120 documents. This makes statistical inferences unreliable. Finally, because we are dealing with software documentation, the terms tend to be rather specific, and their common ancestors are less likely to appear within the document set. We hypothesized that the higher level relationships cut across branches of a 'virtual hierarchy'. This is consistent with the earlier observation that, from a conceptual scope point of view, the concepts we need to describe software components tend to be at the lowest levels of the ACM classification structure, or even lower. This means that, potentially, most of the second level relationships are invalid since the software documentation is not likely to contain general computer science terms, or if it does, those will appear infrequently. Table 3 shows a level by level breakdown of relationships. The overall degra-

[5] If a term occurred only a handful of times, its ENTROPY will be close to the maximum even in those cases where it identifies a narrow subset of documents. F_o is the overall frequency over which we start 'demanding' focussed occurrences. We used thresholds that were close to 1/5th the size of the document set, which here means around 24.

Table 3
Distribution of links across the levels of the hierarchy

	No. of terms	Percentage of invalid terms	Percentage of has-broader-term	Percentage of related	Percentage of unrelated
Level 2	10	10.0	20.0	60.0	10.0
Level 3	24	0.0	37.5	50.0	12.5
Level 4	36	13.0	22.0	47.0	16.67
Level 5	53	9.0	21.0	41.0	28.0
Level 6	65	6.0	20.0	35.0	38.0
Level 7	47	6.0	17.0	30.0	46.0
Level 8	28	11.0	21.0	29.0	39.0
Level 9	14	14.0	0.0	36.0	50.0
Level 10	6	0	16.67	16.67	66.67

dation of the quality of the links within the hierarchy as we go down is consistent with the unreliability of the results for the less frequent terms; we cannot make much of the fact that the level links are of a lesser quality than the level 3 terms, but the above hypothesis is worth exploring.

We considered merging the resulting hierarchy with the ACM hierarchy (see e.g. Ref. [19]) whereby, if a term *T* appears in both ACM and the automatically generated hierarchy, we carry over the subtree from the automatically generated tree to the ACM subtree. We found only eight such common terms between the ACM tree (1200 + nodes) and the automatically generated hierarchy (190 + nodes).

We made several other refinements that improved the quality of the hierarchy only marginally, if at all. Ways to improve the results include using larger data sets in general, but also using a document collection that covers a broad spectrum of conceptual depth and precision. For the purposes of the retrieval experiment, the automatically generated hierarchy was used as a flat set of terms, since we could not rely on the quality of relationships.

4. Automatic indexing from controlled vocabulary

4.1. The algorithm

Traditionally, controlled-vocabulary indexing is done manually, which is a labor-intensive task. We attempted to automate it, at the cost of losing *some*, but hopefully not *all* of the advantages of controlled vocabulary indexing. Simply put, our approach works as follows: a document *D* is assigned a term $T = w_1 w_2 \cdots, w_n$ if it contains (most of) its component words, consecutively ('$\ldots w_1 w_2 \cdots w_n \ldots$'), or in close proximity ('$\ldots w_1 n_1 n_2 w_2 w_3 \cdots w_n \ldots$'). In our implementation, we reduced the words of both the terms of the vocabulary and the documents to their word stem by removing suffixes and word endings. Also, we used two tunable parameters for indexing, (1) proximity, and (2) threshold for the fraction of the number of words found in a document, to the total number of words of a term; a term was assigned if that fraction is above the threshold. Assume that the vocabulary contains the term (key phrase) Database Management Systems. A threshold of 2/3 would assign the term to any document that contained two or more words out of three. The proximity parameter indicates how many words apart should words appear to be considered part of the same noun phrase (term). Maarek et al. had found that five worked well for two-word phrases in English [16]. It has been our experience that indexing works best when both parameters depend on the size of the term. A threshold that is an increasing function of the number of words in a term seems to yield a balanced mix of short and long terms, with reasonably few false-positive assignments. Similarly, what seems to work best for proximity is to use an *m*-word distance between any two neighboring words, but a smaller overall spread than $n \times m$, where n is the number of words in the term.

At first glance, this approach seems to suffer from similar problems to automatic plain-text indexing because of its potential for false positives—still matching words regardless of semantic context—and false negatives—still relying on the terminology used by technical writers or developers. We felt, however, that because we are dealing mostly with compound terms, the proximity and threshold parameters provide both some context for the matching, thereby reducing the chances of false positives, and some flexibility in matching, thereby reducing the chances of false negatives. Further, notwithstanding the quality of indexing, the fact that searchers are constrained to use the same vocabulary that was used for indexing can eliminate a good many sources of retrieval errors.

4.2. Results

The indexing algorithm was used to index the Description attribute of the library components. In particular, we used a threshold of 2/3 and a proximity of 5, meaning that we assign a term when at least two thirds of the words of the term occurred in the textual part of the attribute, with no two words more than five words apart. The results of the indexing were somewhat difficult to analyze directly because the quality of indexing is related as much to the quality of the vocabulary as it is to the indexing algorithm. For example, we know that names of classes, methods, or variables should not have been included in the indexing vocabulary in the first place—about 26 terms. Another factor came into play: terms that make sense in the context of other terms, make little sense when taken alone. For example, the hierarchy contained the path 'Size' → 'Allocation' → 'Block of Memory', and one intuitively reads Size as Size of Block of Memory or Size of Allocation of Block of Memory. Suppose, however, that the term Size alone were assigned to the description of a component; it means very little in this context. This problem is not unique to the automatically generated hierarchy: the ACM Computing Reviews classification structure has several instances of nodes which should be 'read' in conjunction with their ancestors to be meaningful.[6]

In order to separate the issue of vocabulary control from the performance of automatic indexing per se, we indexed the in-line textual documentation of classes with the ApplicationDomain vocabulary. While we did not expect to find the same term assignment as the manual indexing, 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 ranging 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 because termin. differences	Missing because words missing
Number of terms	42	3	6	11	24
Percentage among assigned	82	6	12		
Coverage	52	4		14	30

The extraneous terms are terms that should not have been assigned (false-positive). Examples include the indexer mistaking the verb [this method] 'sets' for the word 'Sets' (as in collections). Some of these cases can be resolved if we combine word matching with part-of-speech tag matching so that names match names, and verbs match verbs. 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 if they had the same text, 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 'verbally' altogether, but is implicit. An example of (a) is the use of the word 'Array' in the text, and the word 'Vector' in the on-line documentation.[7] Examples of (b) include the sentence 'matches upper case character' missing the term 'Pattern Matching' or 'String comparison'. It also includes a number of cases where a term is a conjunction as in 'Strings and Symbols', and only one of the two words appearing in the text, coming short of the 2/3 threshold. This happened quite a few times, and can be easily resolved by tagging conjunctive terms to tell the indexer to assign the whole term if it matches one or the other. This may involve, among other things, rewriting terms such as 'Information Storage and Retrieval' as '(Information Storage) and (Information Retrieval)'.

In summary, only 6% of the assigned terms were wrong, which should only minimally affect retrieval precision. However, the indexer seems to have missed a significant number of terms (44%), although that number can be reduced using minor refinements. We cannot estimate what the effect of these 'false-negative' term assignments will be on retrieval recall. For instance, on any given document or component, the effect of removing an index term on the retrievability of the document or component will depend on the other terms already assigned (are there any, are they related to the removed term), and on the retrieval algorithm used (does it use exact retrieval, does it measure 'conceptual distance' between related terms, etc).

5. Retrieval experiments

5.1. Experimental design

We were as concerned with establishing the usefulness of the library tool in a production setting as we were with performing comparisons between the various retrieval methods. It is our belief that such comparisons do not mean much if a developer will not use ANY of the methods in a real production setting. The decision for a developer to use or not use a tool has to do with, (1) his/her estimate of the effort it takes to build the components from scratch [35], (2) the cost of using the library tool, including formulating the queries and looking at the results, and (3) the perceived track record of the tool and the library in terms of either finding the right components, or quickly 'convincing' the developer that none could be found that satisfy the query. By contrast, comparative studies between the retrieval methods focus on the retrieval performance, regardless of the cost factors. Further, to obtain a fair and finely detailed comparison, the format of the queries is often restricted in those experiments to reduce the number of variables, to the point that they no longer reflect normal usage of the library.

With these considerations in mind, we made the following choices:

(1) We only controlled the search method that the users could use to answer each of the queries, without giving a time limit on each query, or a limit on

[6] Such terms are sometimes called *minor descriptors*, i.e. property names attached to their parent concepts; obviously the property name alone does not mean much as several concepts may share the same property.

[7] This is an interesting discrepancy because it illustrates a fundamental difficulty in software component retrieval. The on-line documentation rightly focuses on abstractions, and hence used the word 'Vector'. The in-line documentation (within program code, like javadoc comments) describes the *implementation*. Developers will be querying based on abstractions, and not on implementations. Actually, ideally, we should let them query based on *problems*, altogether, but that is another story.

the number of trials made for each query; we assumed that users will stop when they are convinced that they have found all that is relevant,

(2) We logged the actions of the subjects with the tool. This provided us with finer experimental data without interfering with the subjects' workflow.

By giving users this much freedom, we run the risk that user bias will skew the data in one direction, preventing us from performing reliable analyses. For example, with boolean retrieval, subjects could search on two search attributes, separately or in combination. Recall that one attribute, Application Domain, was indexed manually with a manually built vocabulary, while the other, Description, was indexed automatically with the automatically generated hierarchy (see Section 4). We did not ask the subjects to use one or the other, or both in combination. When we studied the traces, it turned out that the Description attribute was used only twice out of a possible 43 keyword queries, and neither query returned a relevant document, which makes any formal comparison of the two attributes impossible. However the fact that the Description attribute was used only twice tells us that subjects did not feel it provided useful information, and that, in and of itself, is a valuable data.

The experimental data set consisted of about 200 classes and 2000 methods from the OSE library. We used 11 queries, whose format is discussed Section 5.2. Seven subjects participated in the experiment, although only the data from 5 subjects was usable. All subjects were experienced C++ programmers. They included two professors, three graduate students, and two professional developers working for the industrial partners of the project. 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. 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. The experiment started with a general presentation of the functionality of the tool set (about 45 *mn*), followed by a hands-on tutorial with the tool set (about 1 h), providing the subjects with an understanding of the theoretical underpinnings of the functionalities, as well as some practical know-how. Before leaving, the subjects were asked to fill out a questionnaire to collect their qualitative appreciation of the tool set.

In order to analyze the results, we used the query questionnaires to compare the subjects' answers to ours, which were based on a thorough study of the library's user manual and some code inspection, where warranted. The log traces provided more detailed information and were used to support finer analyses.

5.2. Queries

Information retrieval systems suffer from the difficulty users have in translating their needs into searchable queries. The issue is one of translating the description of a problem (their needs) into a description of the solution (relevant documents). With *document* retrieval systems, problems may be stated as 'I need to know more about ⟨X⟩', and solutions as 'A document that talks about ⟨Y⟩'. For a given problem, the challenge is one of making sure that ⟨X⟩ and ⟨Y⟩ are the same, and in systems that use controlled vocabulary indexing, trained librarians interact with naive users to help them use the proper search terms.

With *software component* retrieval, the gap between problem statement (a requirement) and solution description (a specification) is not only terminological, but also conceptual. In an effort to minimize the effect of the expertise of subjects in an application, and their familiarity with a given library, controlled experiments in component retrieval usually use queries that correspond closely to component specifications. This does not reflect normal usage for a reusable components library tool. For instance, users typically do not know how the solution to their problem is structured, and for the case of a C++ component library, e.g. the answer could be a class, a method, a function, or any combination thereof. It has generally been observed that developers need to know the underlying structure or architecture of a library to search for components effectively [22]. Accordingly, in an effort to get a realistic experiment, we formulated our queries as problems to be solved. Each query was preceded by a problem description setting up the context, followed by a statement 'Find *a way* of ⟨performing a given task⟩'. The problem description is also used to familiarize the subjects with the terminology of the application domain using textbook-like language.

5.3. Component relevance: a performance-based evaluation

The difference between traditional bibliographic document retrieval and reusable component retrieval manifests itself in the retrieval evaluation process as well. The concept of relevance, which serves as the basis for recall and precision measures, is notoriously difficult to define. With bibliographic document retrieval, a search query for a concept X is understood as meaning 'I want documents that talk about X', and hence, a document is relevant if it 'talks about' X. This definition is different from pertinence which reflects a document's usefulness to the user [30]. The usefulness of a document to the user depends, among other things, on the user's prior knowledge, or on the pertinence of the other documents shown to them. Recall, which measures the number of relevant documents returned by a query to the total number of relevant documents in the document set, implicitly assumes that all the relevant documents are equally pertinent and irreplaceable: the user needs all of them. In other words, with traditional document retrieval, assuming that a query Q has N relevant documents, and retrieved a set of documents $S = \{D_1, \ldots, D_m\}$, we can define *pertinence*, and recall as follows:

$$\mathrm{PERT}(D_i) = \begin{cases} \dfrac{1}{N}, & \text{if } D_i \text{ is relevant} \\ 0, & \text{if } D_i \text{ is not relevant} \end{cases} \quad \text{and } \mathrm{PERT}(S)$$

$$= \mathrm{RECALL}(S) = \sum_{j=1}^{m} \mathrm{PERT}(D_j)$$

With software component retrieval, the notions of pertinence (usefulness) and substitutability are much easier to define as both relate to a developer's ability to solve a problem with the components at hand. Symbolically, we view query as a requirement Q, which may be satisfied by several, possibly overlapping, sets of components $S_1, \ldots, S_k$, where $S_i = \{(D_{i_1}, D_{i_2}, \ldots, D_{i_k}\}$. As a first approximation, we define as follows:

$$\mathrm{PERT}(S_i) = \mathrm{PERT}(D_{i_1}, D_{i_2}, \ldots, D_{i_{k_j}}) = \sum_{j=1}^{k_i} \mathrm{PERT}(D_{i_j}/S_i) = 1 \qquad (2)$$

where $\mathrm{PERT}(D/S_i)$ is the usefulness or pertinence of the component D in the context of the solution set S_i. This illustrates the fact that a retrieved component D is useful 'only if' the other components required to build a solution are retrieved with it. Further, this definition of PERT means that total user satisfaction can be achieved with a subset of the set of relevant components, which is not the case for recall. We illustrate the properties of PERT through an example.

Consider two solutions sets $S_1 = \{D_1, D_2\}$ and $S_2 = \{D_1, D_3, D_4\}$ and assume that D_1, D_2 and D_3 have the sizes 30, 20, 40, and 30, respectively, giving S_1 and S_2 the sizes 50, and 100, respectively. We can use the relative sizes of the components with respect to the enclosing solution as their contextual/conditional pertinence, i.e. $\mathrm{PERT}(D_i/S_j) = \mathrm{size}(D_i)/\mathrm{size}(S_j)$. In this case $\mathrm{PERT}(D_1/S_1) = 0.6$, $\mathrm{PERT}(D_2/S_1) = 0.4$, $\mathrm{PERT}(D_1/S_2) = 0.3$, $\mathrm{PERT}(D_3/S_2) = 0.4$, and $\mathrm{PERT}(D_4/S_2) = 0.3$. Assume that a query retrieves the component D_1. In this case, $\mathrm{PERT}(D_1) = \mathrm{Max}(\mathrm{PERT}(D_1/S_1), \mathrm{PERT}(D_1/S_2)) = 0.6$. If the query retrieved D_1 and D_3, instead, $\mathrm{PERT}(\{D_1, D_3\}) = \mathrm{Max}(\mathrm{PERT}(\{D_1, D_3\}/S_1), \mathrm{PERT}(\{D_1, D_3\}/S_2)) = \mathrm{Max}(\mathrm{PERT}(D_1/S_1) + \mathrm{PERT}(D_3/S_1), \mathrm{PERT}(D_1/S_2) + \mathrm{PERT}(D_3/S_2)) = \mathrm{Max}(0.6 + 0.0, 0.3 + 0.4) = 0.7$. This illustrates the fact that when several partial solutions are returned by the system, we take into account the one that is most complete, and the value of individual components is relative to that solution. Symbolically, given the solution sets $S_i, \ldots, S_k$, a query that returns a set of components S has the pertinence:

$$\mathrm{PERT}(S) = \underset{j=1,\ldots,k}{\mathrm{Max}}\ \mathrm{PERT}(S \cap S_j/S_j) \qquad (3)$$

Finally, we add another refinement which takes into account the overlap of two components within the same solution set. Consider the solution S_1 above, and assume that the system retrieves D_1 and D_2', where D_2' is *a superclass* of D_2 that implements only part of the functionality required of D_2. In this case, we could take $\mathrm{PERT}(D_1/D_2') = 0.6 + 0.3 = 0.9$. If the query retrieved D_2' *AND* D_2, then we discard the weaker component. This is similar to viewing solutions sets as role fillers and, for each role, take the component that most closely matches the role. Within the context of reusable OO components, *roles* may be seen as class interfaces, and role fillers as class implementations.

For our experiments, some of the 11 queries were straightforward in the sense that there was a single component (a method or a class) that answered the query, and both component relevance and recall were straightforward to compute. Queries whose answers involved several classes collaborating together (e.g. an object framework) were more complex to evaluate and involved all of the refinements discussed above.

For the case of precision, we used the traditional measure, i.e. the ratio of the retrieved components that were relevant (i.e. had a non-zero $\mathrm{PERT}(\cdot)$) to the total number of retrieved components. We can also imagine refining the definition of precision to take into account the effective usefulness of the individual components, and factor that in with the cost of retrieving and examining a useless component. The cost of examining a useless component is a function of its complexity, and size could be used as a very first approximation of that complexity.

5.4. Performance results

Table 4 shows recall and precision for the 11 queries.

For each query, we randomly selected three subjects out of the initial seven to perform the query using full-text retrieval, and the remaining four subjects to perform keyword retrieval, or vice versa, while making sure that

Table 4
Summary of retrieval results

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

each subject had a balanced load of full-text and keyword queries (6 and 5, respectively, or vice-versa). Because the results of two subjects could not be used, we ended up with some queries answered by four subjects using full-text retrieval, say, and only once using keyword retrieval (see e.g. query 2). The 11th query was rejected because the three keyword-based answers were all rejected for one reason or another. Hence, comparisons between the two methods for the individual queries are not reliable.

At first glance, it appears that plain-text retrieval yielded significantly better recall and somewhat better precision. It also appears that it has done consistently so for the 10 queries, with a couple of exception. In order to validate these two results statistically, we have to ascertain that none of this happened by chance. We performed a number of ANOVA tests, to check whether recall and precision were random variables of the pair (query, search method), and both tests were rejected. Next, we isolated the effect of the search type to see if the difference in recall and precision performance is significant. The results are shown in Table 5.

The '$\Pr > F$' shows the probability that such a difference in performance could have been obtained by chance. It is generally accepted that a threshold of 5 percent is required to affirm that the differences are significant. Thus, we conclude that:

- Full-text retrieval yields provably/significantly better recall than controlled vocabulary-based retrieval
- Full-text retrieval yields comparable precision performance to that of controlled vocabulary-based retrieval.

Our results seem to run counter to the available experimental evidence. Document retrieval experiments have consistently shown that controlled vocabulary-based indexing and retrieval yielded better recall and precision than plain-text search [4,30,31], although the difference was judged by many as being too small to justify the extra costs involved in controlled vocabulary-based indexing and retrieval [31]. Similarly, a comparative retrieval experiment for reusable components conducted by Frakes and Pole[8] at the SPC showed that recall values were comparable, and a superior precision for controlled vocabulary-based retrieval [8]. Most surprising in our results is the significant difference is recall performance. We analyze these results in more detail below.

To explain these results, we formulated and tested a number of hypotheses. We first note that out of the 11 queries, some were supposed to retrieve single components

Table 5
Significance of differences between plain-text retrieval and keyword retrieval

Effect of search method	Recall	Precision
F value	4.1	0.93
$\Pr > F$	0.0500	0.3404

(often methods), as in Query 7, formulated as 'getting the length of a string', and the others were supposed to retrieve a collection of components with complex interactions, often a mix of classes and methods. With full-text search, queries retrieve indiscriminately methods and classes. With controlled-vocabulary search, users have to instantiate different query templates, depending on the kind of components they are seeking (a class or a method). We hypothesize that this makes the search more tedious and users may give up search easily, yielding lower recall. For this hypothesis to hold, there has to be a marked difference between the performance for the single-component queries (queries 1, 7, 8, 9) and the queries whose answers consisted of collections of components (queries 2, 3, 4, 5, 6, 10). Table 6 compares the two kinds of queries.

Our hypothesis that plain-text retrieval favors component collection queries is not validated. Along the same lines, we hypothesized that plain-text retrieval favored queries whose answers involved a mix of methods and classes, or just classes, since the same query would retrieve both kinds of components. Table 7 shows recall and precision values for the two retrieval methods, separated into the two kinds of queries.

This hypothesis is not validated: in both cases, plain-text retrieval is markedly superior to controlled-vocabulary retrieval with regard to recall—and somewhat with regard to precision for the case of queries whose answers included both classes and methods. Note, however, that there is a marked difference in performance between the two groups of queries.

Could the quality of indexing be to blame for the lower performance of controlled-vocabulary based retrieval? Recall that we indexed two attributes, Application Domain and Description. The Application Domain attribute was indexed manually and fairly systematically, thanks to the quality of on-line documentation. There are two potential weakness of this indexing, but none can account for the observed difference in performance

Table 6
Comparing the two sets of queries

Query set	Full-text retrieval		Keyword retrieval	
	% Recall	% Precision	% Recall	% Precision
Single comp. queries	62.08	84.67	44.08	65.333
Comp. coll. queries	64.43	68.17	41.30	70.33

[8] Frakes and Pole compared four methods, and their test of statistical significance was based on variance analysis of the precision averages for the four methods, which was inconclusive [8]. However, we are quasi-certain that by performing pairwise comparison between plain-text search (50%) and controlled vocabulary search (what appears to be 100% on the plot [8]), they would have established, statistically, the superiority of controlled-vocabulary retrieval.

Table 7
Comparing the two sets of queries depending on whether they retrieve methods or not

Query set	Full-text retrieval		Keyword retrieval	
	% Recall	% Precision	% Recall	% Precision
Answer = classes and methods	41.333	59.17	27.77	47.27
Answer = classes only	85.65	89.77	57.05	89.40

between the two retrieval methods. First, some methods were left unindexed because the on-line documentation said nothing about these methods—such as constructors. The results of Table 7 bear this out: keyword retrieval performed better on queries whose answers involved only classes than on queries whose answers involved a combination of classes of methods. However, this does not explain the fact that free-text retrieval performed better than keyword retrieval *for both types of queries*. The second potential weakness of the indexing of the ApplicationDomain is the fact that index terms are sometimes perceived as too general. Indexing that is too general results in poor precision, but is known to produce *better* recall, which is not what we observed.

The attribute Description was indexed automatically (see Section 4) with the vocabulary that was generated automatically (see Section 3). Notwithstanding the quality of indexing of this attribute, the experiment logs showed that this attribute was actually used only three times, and in all three cases, it was used in conjunction with ApplicationDomain, but failed to match any component. Accordingly, even in those cases where it was used, it did not affect the ranking of components returned by weighted boolean retrieval (see Section 2.3.2). The fact that the attribute Description was not used as often as ApplicationDomain could be explained by the nature of queries: the queries were presented as programming problems or tasks to solve, rather than a look-up for components given a set of specifications. Because ApplicationDomain talks about problems that components help solve whereas Description talks about how these components are implemented, it makes sense that the former be used more often than the latter in the queries.

We continue to analyze the results of this experiment, as the logs provide us with a wealth of information and hypotheses that we could validate. We do not expect this experiment to reverse the long-held consensus that controlled vocabulary performs better than free-text retrieval; more experiments that target narrower retrieval tasks, and that involve fewer operational parameters would be needed for that. We can view it in light of another emerging consensus according to which, whichever performance benefits controlled vocabulary indexing and retrieval might have—in our case none, quite the contrary—they hardly justify the added cost. Perhaps more importantly, four subjects out of five preferred plain-text search.

More importantly, we believe that this experiment contributes to a needed rethinking of reusable component retrieval paradigms and tools. Such implications are discussed next.

6. Conclusion and directions

We set out to develop, evaluate, and compare two classes of component retrieval methods which, supposedly, strike different balances along the costs/benefits spectrum, namely, the (quasi-) zero-investment free text classification and retrieval versus the ‘up-front investment-laden’ but presumably superior controlled vocabulary faceted indexing and retrieval. Recent experiments with software component repositories have put into question the *cost-effectiveness* of the controlled vocabulary approach, but not its *superior* or *at least equally good* retrieval performance [8]. We attempted to bring the two kinds of methods to a level-playing field by: (1) addressing the costs issue by automating as much as possible of the pre-processing involved in controlled vocabulary-based methods, and (2) using a realistic experimental setting and realistic evaluation measures. 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.

Because these results are somewhat counter-intuitive, we continue to analyze them, along with the log data, and to design new experiments that are better targeted towards validating the various hypotheses discussed in Section 5.4. However, they give legitimacy and some urgency to some of the questions we and others have raised about the retrieval of reusable software components [22,23,36].

From an organizational issues point of view, there was wide recognition in the late eighties that reuse will not happen at a large scale within organizations without the proper structuring and management. It was possible, in that context, to conceive of centralized reuse repositories with well-defined roles and quality control criteria and mechanisms [25]. Nowadays, a lot more reuse happens in the unstructured and decentralized world of the Internet and open source software, and any ‘virtual reuse repository’ can only rely on *automated indexing and retrieval methods, regardless of differences in performance*.

Reuse repositories are also facing a number of *paradigmatic* issues. First, there exist qualitative differences between bibliographic document retrieval and software component retrieval [22], which make some of the document retrieval analogies inappropriate. Document library users who do not find the documents they are looking for will look even harder because they cannot perform the tasks for which they needed the information otherwise. A software developer will more easily give up and get on with developing the software component from scratch. As reuse repository designers, we need to account for the fact that software developers are not our captive users, which puts more pressure on us to provide more useful and less intrusive tools. It is important that the use of the repository integrates well into the workflow of developers; this has led some people to suggest that reuse repositories should be active in the sense of presenting potentially relevant information to users before they ask for it [36]. It also means that issues of usability are paramount; if developers prefer a particular search method, then that is the one we should focus on. Our tool set does not address this issue specifically, but we take seriously the fact that four out of five users preferred free text search, which confirms earlier studies. In our case, it even performed better.

Surely, our experiments suggest that there is ample room for improvement in several areas (see Sections 3.2.3, 4.2, and 5.4). However, we believe that there is something more fundamental at play. We believe 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 coincides with analysis, and is fairly exploratory, as developers do not yet know which form (specification?) the solution to their problem will take. During this stage, a free-format search technique such as plain-text search is appropriate, as multi-faceted search may be too rigid and constraining. After contemplating several designs, a developer may then start searching for components that would play a given role within a design, and multi-faceted classification may be too poor for this stage. The format of our queries (problems to be solved), and the fact that experimental subjects used mostly the ApplicationDomain attribute, setting aside the more implementation-oriented Description attribute seem to point in this direction. A combination of free-text search and active reuse repositories [36] may be worth exploring.

Acknowledgements

This work was supported by grants from Canada's Natural Sciences and Engineering Research Council (NSERC), TANDEM Computers, Québec's *Fonds pour la Création et l'Aide á la Recherche* (FCAR), and Québec's *Ministére de l'Enseignement Supérieur et de la Science* (MESS) under the IGLOO project organized by the Centre de Recherche Informatique de Montréal.

Bertrand Fournier, a statistician with the *Service de Consultation en Analyse de Données* (SCAD, http://www.scad.uqam.ca), and Professor Manzour Ahmad, director of SCAD, provided us with invaluable assistance in measuring and interpreting the results.

References

[1] ACM, An introduction to the CR classification system, Computing Reviews January (1985) 45–57.

[2] P. Allen, Reuse in the component marketplace, Component Development Strategies 11 (8) (2001).

[3] H. Bilofsky, C. Burks, J.W. Fickett, W.B. Goad, F.I. Lewitter, W.P. Rindone, C.D. Swindell, C. Tung, The GenBank genetic sequence databank, Nucleic Acids Research 14 (1986) 1–4.

[4] D. Blair, M.E. Maron, An evaluation of retrieval effectiveness for a full-text document-retrieval system, Communications of the Association for Computing Machinery 28 (3) (1985) 289–299.

[5] D. Cutting, J. Kupiec, J. Pedersen, P. Sibun, A practical part-of-speech tagger, Proceedings of the Applied Natural Language Processing Conference (1992).

[6] E. Damiani, M.G. Fugini, C. Bellettini, A hierarchy-aware approach to faceted classification of object-oriented components, ACM Transactions on Software Engineering and Methodology 8 (3) (1999) 215–262.

[7] G. Dumpleton, OSE—C++ Library User Guide, Dumpleton Software Consulting Pty Limited, Parramatta, 2124, New South Wales, Australia, 1994, 124 pp.

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

[9] W.B. Frakes, R. Baeza-Yates, Information Retrieval: Data Structures and Algorithms, Prentice-Hall, Englewood Cliffs, NJ, 1994.

[10] R.J. Hall, Generalized Behavior-based Retrieval, Proceedings of the 15th International Conference on Software Engineering, ACM Press, Baltimore, MD, 1993, pp. 371–380.

[11] S. Henninger, Using iterative refinement to find reusable software, IEEE Software 11 (5) (1994) 48–59.

[12] S. Isoda, Experience report on a software reuse project: its structure, activities, and statistical results, Proceedings of the 14th International Conference on Software Engineering, Melbourne, Australia May (1992) 320–326.

[13] I. Jacobson, M. Griss, P. Jonsson, Software Reuse: Architecture, Process and Organization for Business Success, Addison-Wesley, Reading, MA, 1997.

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

[15] M.E. Lesk, Word–word associations in document retrieval systems, American Documentation 20 (1) (1969) 27–38.

[16] Y.S. Maarek, D.M. Berry, G.E. Kaiser, An information retrieval approach for automatically constructing software libraries, IEEE Transactions on Software Engineering 17 (8) (1991) 800–813.

[17] A. Mili, R. Mili, R. Mittermeir, Storing and retrieving software components: a refinement-based approach, Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy May (1994).

[18] H. Mili, R. Rada, Building a knowledge base for information retrieval, Proceedings of the Third Annual Expert Systems in Government Conference October (1987) 12–18.

[19] H. Mili, R. Rada, Merging Thesauri: principles and evaluation, IEEE Transactions on Pattern Analysis and Machine Intelligence 10 (2) (1988) 204–220.

[20] H. Mili, R. Rada, Medical expertext as regularity in semantic nets, Artificial Intelligence in Medicine 2 (1990) 217–229. Elsevier Science Publishers.

[21] H. Mili, R. Rada, W. Wang, K. Strickland, C. Boldyreff, L. Olsen, J. Witt, J. Heger, W. Scherr, P. Elzer, Practitioner and SoftClass: a comparative study of two software reuse research projects, Journal of Systems and Software 27 (1994).

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

[23] H. Mili, E. Ah-Ki, R. Godin, H. Mcheick, Another nail to the coffin of faceted controlled-vocabulary component classification and retrieval, Proceedings of the '97 Symposium on Software Reuse, Boston, MA May (1997) 89–98.

[24] H. Mili, H. Sahraoui, Describing and using frameworks, in: R.E. Johnson (Ed.), Building Application Frameworks: Object-oriented Foundations of Framework Design, Wiley, New York, 1999, pp. 523–561.

[25] H. Mili, A. Mili, S. Yacoub, E. Addy, Reuse-based Software Engineering: Techniques, Organization, and Control, Wiley, New York, 2002, ISBN 0-471-39819-5.

[26] OASIS, Business Process, Business Information Analysis Overview (ebXML), Organization for the Advancement of Structured Information Standards, May 11, 2001, http://www.ebxml.org/specs/bpOVER.pdf.

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

[28] R. Prieto-Diaz, P. Freeman, Classifying software for reusability, IEEE Software January (1987) 6–16.

[29] R. Rada, H. Mili, E. Bicknell, M. Blettner, Development and application of a metric on semantic nets, IEEE Transactions on Systems, Man, and Cybernetics 19 (1) (1989) 17–30.

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

[31] G. Salton, Another look at automatic text-retrieval systems, Communications of the Association of Computing Machinery 29 (7) (1986) 648–656.

[32] C. Smith, MEDLINE Queries and Distances in MeSH, Internal Report, National Library of Medicine, 1985.

[33] D. Soergel, Organizing Information: Principles of Data Base and Retrieval Systems, Academic Press, Orlando, FL, 1985.

[34] B.H. Weinberg, J.A. Cunningham, The relationship between term specificity in MeSH and online postings in MEDLINE, Bulletin Medical Library Association 73 (4) (1985) 365–372.

[35] S.N. Woodfield, D.W. Embley, D.T. Scott, Can programmers reuse software, IEEE Software July (1987) 52–59.

[36] Y. Ye, G. Fischer, Promoting Reuse with Active Reuse Repository Systems, Proceedings of the Sixth International Conference on Software Reuse, Lecture Notes in Computer Science, vol. 1844, Springer, Berlin, 2000, pp. 302–317.

[37] A.M. Zaremski, J.M. Wing, Signature matching: a key to reuse, Software Engineering Notes 18 (5) (1993) 182–190. First ACM SIGSOFT Symposium on the Foundations of Software Engineering.

[38] A.M. Zaremski, J.M. Wing, Specification matching: a key to reuse, Software Engineering Notes 21 (5) (1995) Third ACM SIGSOFT Symposium on the Foundations of Software Engineering.

[39] G.K. Zipf, The Psycho-Biology of Language, MIT Press, Cambridge, MA, 1965.