Another nail to the coffin of faceted controlled-vocabulary component classification and retrieval

Hafedh Mili, Estelle Ah-Ki, Robert Godin, and Hamid Mcheick
Département d'Informatique
Université du Québec à Montréal
Case Postale 8888 (A)
Montréal, PQ H3C 3P8, CANADA

Abstract

Our research centers around exploring methodologies for developing reusable software, and developing methods and tools for building *with* reusable software. In this paper, we focus on reusable software component retrieval methods that were developed and tested in the context of *ClassServer*, an experimental library tool developed at the University of Québec at Montréal to explore issues in software reuse [15]. The methods discussed in this paper fall into two categories, 1) string search-based retrieval methods, and 2) keyword-based retrieval methods. Both kinds of methods have been implemented and tested by researchers, both in the context of software repositories (see e.g. [6, 9]) and in the context of more traditional document libraries (see e.g. [2, 25]). Experiments have shown that keyword-based methods, which require some manual, labor-intensive pre-processing, performed only marginally better than the entirely mechanical string-search methods (see e.g.[6, 25]), raising the issue of cost-effectiveness of keyword-based methods as compared to string search based methods. In this paper, we describe an implementation and experiments which attempt to bring the two kinds of methods to a level-playing field by: 1) automating as much of the pre-processing involved in controlled vocabulary-based methods as possible to address the costs issue, and 2) using a realistic experimental setting in which queries consist of problem statements rather than component specifications, in which query results are aggregated over several trials, and in which recall measures take into account overlapping components. Our experiments showed that string search based methods performed better than semi-controlled vocabulary-based methods, which goes further in the direction of more recent component retrieval experiments which challenged the superiority of controlled vocabulary based classification and retrieval of components (see e.g. [6]).

1. Introduction

The problem of component retrieval has been widely addressed in the software reuse literature. A wide range of component categorization and searching methods have been proposed, from the simple string search (see e.g. [15]), to faceted classification and retrieval (e.g. [21]) to signature matching (see e.g. [27]) to behavioral matching (see e.g. [10, 28] or even [8]). 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 [17]. The cost of implementing a retrieval method involves both initial set-up costs, and the cost associated with formulating, executing and refining queries. In this paper, we describe the implementation and experimental comparison of two such methods, free-text based retrieval and multi-faceted classification and retrieval of reusable code components.

Typically, retrieval experiments focus on abstract performance measures such as *recall* and *precision*, which are used both as absolute measures, or as a way of comparing methods (see e.g. [6]). Recall and precision, which have traditionally been used to measure the performance of bibliographic retrieval systems, have been criticized because they view relevance as a yes/no property, and because they don't take into account the specific goal that the searcher is trying to achieve.

SSR '97 MA,USA

Further, reuse being essentially a cost issue, comparing classification and retrieval methods based on retrieval performance measures alone is of little use for methods whose set-up and use costs are significantly different. Finally, for a reuse library tool to be successful, the cost of reusing has to be perceived by potential reusers as being significantly less than that of developing from scratch [26], and the cost of performing searches is only one of several costs associated with an instance of reuse [17]. These issues have been a major concern of our research, both in the design of retrieval algorithms, and in evaluating them.

Within the context of our research, we have developed four classes of retrieval algorithms: 1) retrieval using full-text search on software documents and program files, 2) navigation through the structure of components, 3) multi-faceted classification and retrieval of components, and 4) signature matching. We are as much interested in performance-based comparisons between the different methods, as we are concerned about getting the tools and methods adopted by organizations and used by their developers in a realistic development environment. Issues of set-up costs for the various methods have been a major concern, and we tried to automate as much of the pre-processing steps as possible, lest we degrade slightly retrieval performances. The experiments described in this paper compare full-text retrieval with a variant of multi-faceted classification in which we attempted to automate some of the labor-intensive pre-processing steps. The experiment design was aimed towards simulating real-life situations, and the evaluation of retrieval performance was goal-oriented, rather than a simple count of returned and potentially useful components. The experiments showed that our efforts at automating multi-faceted classification of components were not very fruitful. They also showed that full-text retrieval of components was superior to multi-faceted retrieval, which contradicts the result

of document retrieval experiments in the information retrieval literature.

In the next section, we describe the representation of software components used by our library tool. In section 3, we describe the multi-faceted retrieval of software components. The retrieval experiments are described in section 4. We conclude in section 5.

2. Representing reusable components

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 (see e.g. [7, 14, 18]), and for developing *with* reusable software (see e.g. [15, 16, 19]). Our work on supporting development *with* reusable components centers around a tool kit called *ClassServer* that consists of various tools for classifying, retrieving, navigating, and presenting reusable components (see Figure 1). Reusable components consist essentially of object-oriented source code components, occasionally with the accompanying textual documentation. The internal representation of reusable components supports the four reuse functionalities mentioned above, namely, classification, retrieval, navigation, and presentation. Raw input source files are put through various tools— called *extractors*— which extract the relevant pieces of information, and package them for the purposes of the four reuse functionalities. The information extracted by these tools includes language-defined structures, such as *classes*, *variables*, *methods*, and *method parameters*. To these, we added *views*, which are client- or application-specific interfaces that classes may support, and the notion of *object frameworks*, which are class-like object aggregates that are used to represent application frameworks and design patterns [20]. Each kind of component is defined by a descriptive template which includes: 1) structural information describing the kind of subcomponents a component can or must 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, which are used for search purposes, as in a *class* has an *author* and an *application domain*, *method* has a *purpose*, etc.

The tool set may be seen as consisting of three subsystems. The first subsystem supports the required functionalities for full-text retrieval of source code *files*. Simply put, words from user queries are matched against the contents of source code files (or other kinds of textual information). Before this matching is done, a number of pre-processing steps are performed with two goals in mind: 1) speeding up search, and 2) maximizing the chances for matching by removing inessential lexical variations. To speed up the search, an *inverted list* is created once and for all, which is a table whose keys are unique "meaningful"[1] words and whose values, for each word, are the documents in which that word occurred. In order to remove inessential lexical variations, the words of the table are first put through a "word stemmer" which reduces a number of word forms to the same "stem" (as in "facility" and "facilities" reducing to "facilit").

The component browser and the keyword retrieval subsystems use the structured representation of the components extracted using the tool referred to as "semantic/structural parser" in the Figure. The parser for C++ was developed using Lex and Yacc, and a public domain C++ grammar [23], which we augmented to handle templates. The Smalltalk parser was written directly in Smalltalk. For both parsers, the parsing produces a trace of the traversal of the abstract syntax stree. The trace consists of a batch of component creation commands, which are executed when we "load" the trace; that is the *structured component loader*.

1. Common words such as "the" "however", etc. are not taken into account in either the inverted list or the queries.

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 [21]. 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 [13]. Filling in the *values* property is referred to as *categorization* or *indexing*. Typically, human experts read about the software component, and chose key words or phrases from a predefined list; this is referred to in the information retrieval literature as *manual controlled-vocabulary indexing* [24]. 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. More on this in section 4.2.2.

For a given attribute, e.g. "Purpose", multiple values are considered to be *alternative* values , rather than *partial* values. For a given vocabulary, the terms of the vocabulary (key words and phrases) may be organized along a conceptual hierarchy. Figure 2 shows excerpts of the conceptual hierarchies of key phrases for the attributes "Application Domain" (Figure 2.a) and "Purpose" (Figure 2.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 "Application Domain" hierarchy of key phrases is inspired from the (ACM) *Computing Reviews*'s classification structure [4]. The hierarchical relationship between key phrases is a loose form of generalization, commonly referred to in information retrieval as "Broader-Term" [24]. Attribute values (key words and phrases) are used in boolean retrieval whereby component attribute values are matched against required attribute values (queries); more on this in § 3. The hierarchical relationships within an indexing vocabulary are used to extend the basic retrieval algorithms by adding different degrees of matching (instead of true or false) between two key words, which now depend of the length of the path separating them in the hierarchy (see [16]).

3. Muti-faceted Retrieval Of Reusable Components

ClassServer includes a search tool-- called *ClassSearcher*-- that enables developers to retrieve software components based on the keyword values of their attributes. We implemented three matching algorithms: 1) a weighted boolean retrieval (see e.g. [24]), 2) conceptual distance measures, and 3) classification. Both conceptual distance measurements and classification use semantic relationships between keywords— *hierarchical* relationships for the case of classification [16]. For the purposes of this paper, we limit our discussion to boolean and weighed boolean retrieval. We begin by discussing the structure of queries and the operations that can be performed on them.

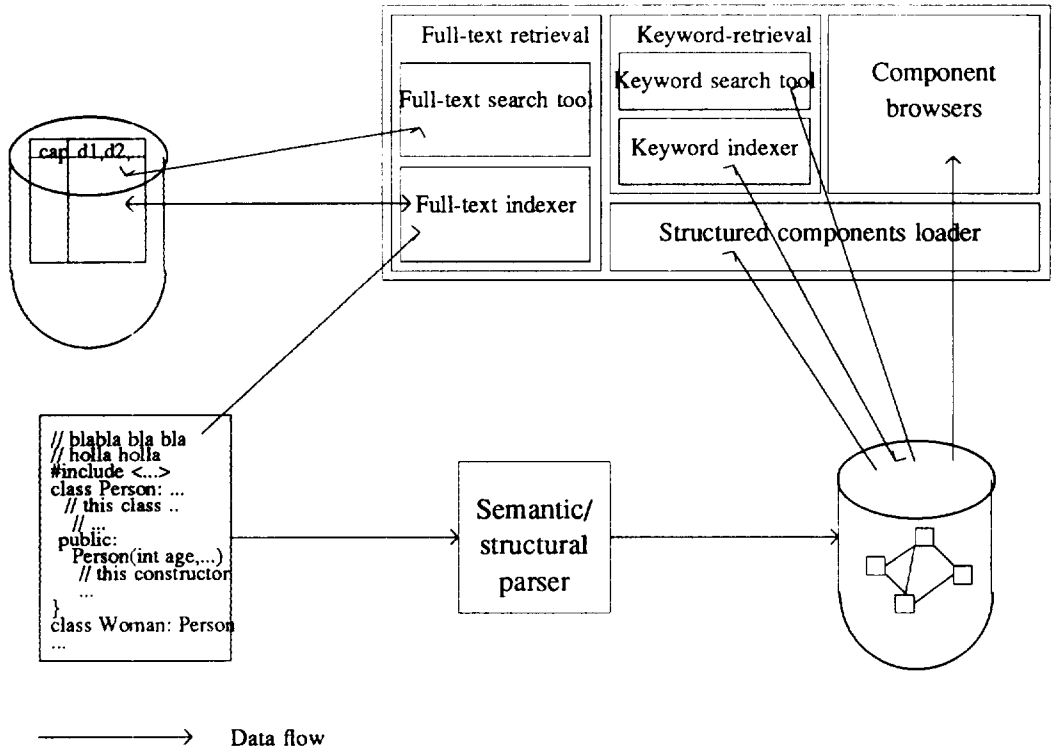

Figure 1. Overall architecture of ClassServer

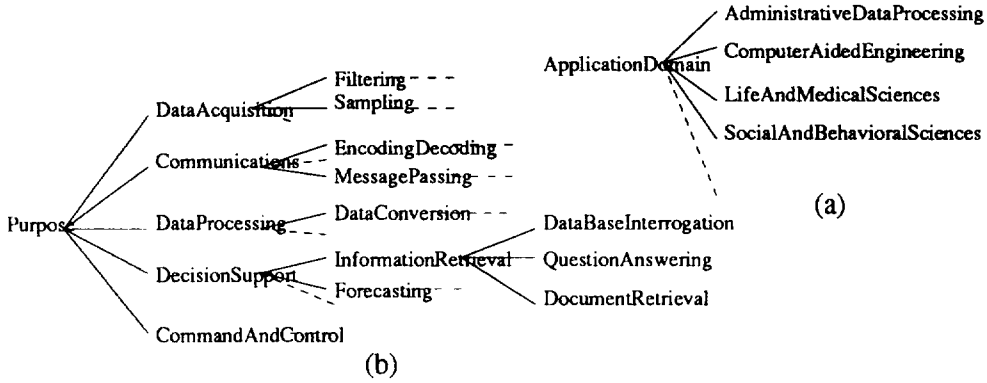

Figure 2. Hierarchies of key phrases for the attributes "Application Domain" and "Purpose".

3.1. Queries

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. As a guiding principle, we likened the specification of a query to the process of specifying a prototypical component. Accordingly, 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, discussed further below, and conceptual distance measures, respectively. 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) $\cap$ ListOfKeyPhrases $\neq \Phi$. The query denoted by the tuple (AQT_1,...,

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,...,AQT_k), where AQT_i = <$Attribute_i$,$Weight_i$,$CutOff_i$,$ListOfKeyPhrases_i$>. The score of a component C is computed as follows:

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

where $Score(AQT_i,C)$ equals 1.0 if $ListOfKeyPhrases_i \cap Attribute_i(C) \neq \Phi$, and 0 otherwise.

Notice that only some combinations of AQTs make sense since each kind of component has a different set of applicable attributes. This, plus our goal of making the specification of queries easy led us to support two ways of initializing queries: 1) by specifying the kind of component we would like to search on, or 2) by specifying a component. In the first case, the query is initialized to an initial list of AQTs corresponding to the default set of attributes for the component category with empty lists of key phrases; the user has then the option of adding and/or removing AQTs, and must specify lists of key phrases for the AQTs. When a query is initialized from a component, in addition to using the attributes of the component to select the initial list of AQTs, we use the values of those attributes to initialize the list of key phrases for those AQTs. We believe that this will actually be the most natural way of submitting a query, in terms of integrating in the workflow of developers: typically, a developer would start entering, explicitly or implicitly, the specifications of a component to be built depending on the problem at hand, and on the component's interactions with other components; those partial specifications may serve as a way of searching, in the library of components, for that that satisfies those requirements.

3.2. Operations On Queries

Queries in ClassSearcher may be combined, *once processed*, using the usual set-theoretic operators (union, intersection, and set difference) directly on the answer lists. This enables developers to formulate arbitrarily complex queries involving negation, ORed AQTs, ANDed values for the same AQT, or combinations thereof. Note, however, that within a single query, the AQTs are *always* ANDed, and multiple values are *always* ORed. This forces us to write queries such as

FIND COMPONENTS C SUCH THAT
[Purpose(C) = Rename $\wedge$ Copy $\wedge$ $\neg$(Move)] OR
[IntendedUsers(C) = UnixHacker]

as a boolean expression of four separate queries. Namely, let Q_1 = ((Purpose,{Rename})), Q_2 = ((Purpose,{Copy})), Q_3 = ((Purpose,{Move})), and Q_4 = ((IntendedUsers,{UnixHacker})), the above request can be satisfied using the boolean expression Q = $(Q_1 \wedge Q_2 \wedge \neg Q_3) \vee Q_4$), or the equivalent expression using set operations: $(Q_1 \cap Q_2 - Q_3) \cup Q_4$.

We chose not to generalize the format of queries for the purposes of accommodating the above query more readily, for a number of practical and theoretical reasons. First, we believe that such queries are relatively uncommon so as not to warrant complicating the format of queries, or offering two different formats. Second, from a theoretical point of view, negation is difficult to express mathematically in the context of graph-based conceptual distance measurements (see [22]). Finally classification is hard to interpret semantically if a query contains two ORed AQTs that correspond to different attributes [12].

3.3. ClassSearcher

Figure 3 shows the graphical interface to ClassSearcher. The upper third of the window contains two identical lists of queries, identified by names. The selection in the left list determines the state of the rest of the interface. The right list is used simply to specify (i.e. select with the mouse) the second operand to the set/logical operations. The pane with heading **Query Transcription** is used for output only, and shows an SQL-like transcription of the currently selected query in the left list; notice that weights and cut-off values are not represented in the transcription.

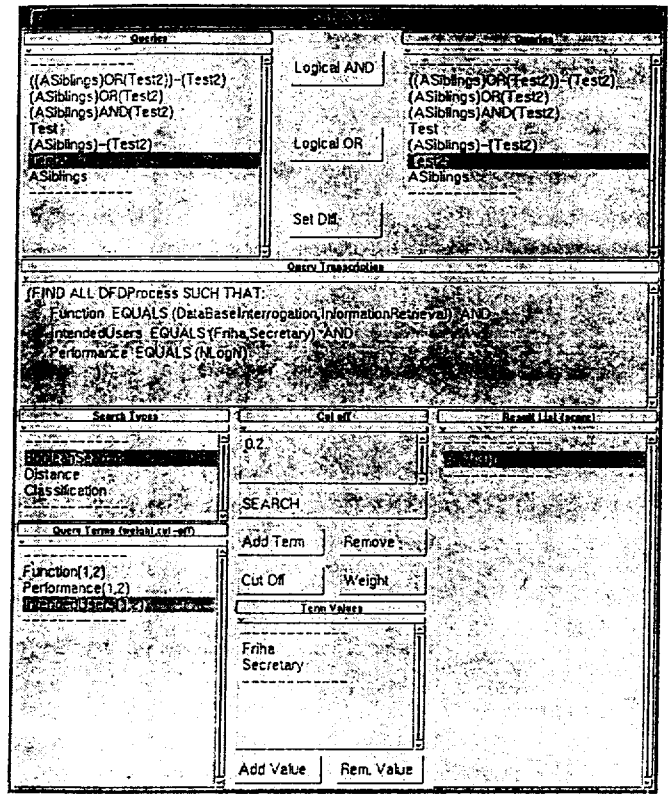

Figure 3. Graphical Interface of ClassSearcher.

The pane labeled **Search Types** is used to set the parameters (cut-off, to the right) and perform (button SEARCH) the currently selected search type— BooleanSearch in this case. **Result List(score)** shows the answer list for the currently selected search type, ordered by decreasing order of score (shown between parentheses). The button "Add Term" is used to add an attribute query term (AQT) to the currently selected query. The list of AQTs for the currently selected query, with

their weight and cut-offs shown between parentheses, is shown in the leftmost bottom pane labeled **Query Terms(weight,cut off)**. The buttons "Remove", "Weight", "Cut Off", and "Add Value" are relative to the currently selected query term (AQT), with self-explanatory names. "Add Value" enables developers to navigate through the hierarchy of keywords of the corresponding attribute (see e.g. Figure 2.a) to select a value. The button "Rem. Value" removes the currently selected value in the list above it (**Term Values**).

4. Retrieval experiments

4.1. Experimental design

In this first set of experiments, we were more concerned with establishing the usefulness of the library tool in a production setting than we were with performing comparisons between the various retrieval methods. It is our belief that such comparisons do not mean anything if a developer won't 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 [26], 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. Typically, comparative studies between the retrieval methods focus on the retrieval performance, regardless of the cost factors. Second, 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 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 "spied" on the subjects' interactions with the tool by recording a trace of the actions performed. This allowed us to obtain finer experimental data without interfering with the subjects' workflow.

Note that the trace data may not always be of sufficient quality to make reliable statistical inferences. For example, with boolean retrieval, subjects could search on two search attributes, separately or in combination. 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 § 4.2.1). We didn't 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 in only two of the 43 keyword queries performed by the different subjects, and neither query returned a relevant document. Accordingly, we have no basis for comparing the quality of the two attributes. However the fact that the "Description" attribute was used only twice tells us that subjects didn't 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 (see § 4.2). We used 11 queries, whose format is discussed in section 4.3. Seven subjects participated in the experiment. 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. Two subjects (a professor and a graduate student) did not complete the experiment, and we had to discard the results of the few queries they DID complete. 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. 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 hour), 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 toolset.

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. For this first experiment, the traces generated for the subjects were analyzed only to determine which attributes were used for boolean keyword retrieval. The traces included enough details, however, to support more and finer analyses.

4.2. The library

As mentioned earlier, the data set consisted of the entire OSE library [5], which contained some 200 classes and 2000 methods distributed across some 230 files. A shell script put the files through a C++ pre-processor (to process the #include directives, among others) before they were input into the C++ extractor, which generated a file of Smalltalk commands to construct the reusable C++ components. Because of the good quality and format consistency of the in-line documentation, we were able to assign C++ comments as text values for the "Description" attribute of various components (classes, methods, variables).

For the purposes of this experiment, we classified components using two attributes "ApplicationDomain", and "Description". Classification according to "ApplicationDomain" was done manually, although fairly systematically. For this "attribute, the classification hierarchy followed closely the table of contents of the textual documentation, whose organization is based on the application areas covered by the library. Generally speaking, if a component is discussed in some paragraph, we assign it the title of the smallest section that contains that paragraph. A consequence of this indexing scheme is that not all methods had values for the attribute "Application Domain". For example, constructor methods were rarely discussed explicitly, and yet, are always present. However, we can be sure that all the classes, or some of their superclasses had values for "Application Domain". For the "Description" attribute, *all* the steps were automated, from the construction of the vocabulary, to the actual indexing of reusable components. Both steps are briefly discussed below. While the quality of the indexing vocabulary and of the indexing method have a significant impact on the quality of retrieval, because the experimental subjects seldom used the attribute "Description" in their queries, we shall keep the

descriptions very brief (see [19] for a thorough discussion); on the other hand, this may very well be the ultimate measure of the quality of that attribute, i.e. whether it was deemed useful or not by the subjects.

4.2.1. Building a hierarchy of domain concepts

There are two aspects to building a conceptual hierarchy. First, there is the problem of finding a set of terms (keywords or key phrases) that describe the important concepts within a domain of discourse, and that use the most widely accepted terminology. To this end, we used a variant of statistical indexing which, given a document collection, it considers as valid content indicators those terms which occur neither too often, nor too rarely, and which occur unevenly in the document collection, i.e. are concentrated in a small subset of the document collection [24]. However, instead of using single words as potential content indicators (or index terms), we used "noun phrases[2]", which we extracted from the OSE software documentation using Xerox's *Parts of Speech Tagger* (XPost) [3].

Having identified the set of important domain concepts, it is important to organize them in a conceptual hierarchy, both for navigation purposes (e.g. to find the proper term to use in a query) and to support the extensions to boolean retrieval mentioned earlier (see also [19]). We adapted an algorithm that we had developed to organize a set of index terms into a graph— more specifically, a hierarchy— based on their usage profiles within a document collection [11]. The basis of that algorithm was the hypothesis that index terms that often characterize the same documents (i.e. co-occur in the documents' indices) tend to be related, and should be connected; additional heuristics were used to refine the relationships [11]. The new adaptation of the algorithm performed rather poorly compared to the experiment described in [11], and this for a variety of reasons, including the more problematic nature of the data, and the much smaller size of the data set [19]. Note however that the quality of the relationships within the generated graph has no impact on indexing (discussed next), and that in the end, it had no measurable impact on retrieval performance because the attribute "Description" was seldom used.

4.2.2. Automatic indexing from a controlled vocabulary

The information retrieval literature makes the distinction between manual controlled vocabulary indexing, and automatic "uncontrolled vocabulary" indexing [24]. With manual controlled vocabulary indexing, subject experts read documents and assign to them an *index* consisting of several terms, meant to be content descriptors, taken from the predefined set of key terms, referred to as the controlled vocabulary. By contrast, the basic premise behind automatic uncontrolled-vocabulary indexing is that the words that *occur* in the document with a certain statistical profile are good content indicators. The relative merits of the two approaches have been thoroughly debated in the literature (see e.g. [2, 25], and we won't indulge into the debate for the purposes of this paper. Suffice it to say that we used a mix of the two methods: we did use a controlled vocabulary for indexing and retrieval, but the indexing was done automatically [19]. At first glance, this approach shares similar problems with automatic plain-text indexing and retrieval to the extent that classification (and retrieval) depends on authors having used the same words as the controlled vocabulary, and on the fact that matches are done in context. We argued in [19] why this is not a big problem here, and that we are able to achieve the advantages of controlled vocabulary indexing at a fraction of the cost.

Simply put, automatic indexing with a controlled vocabulary works as follows: a document D is assigned a term $T=w_1w_2 \cdots w_n$ if it contains (most of) its component words, consecutively ("... $w_1w_2 \cdots w_n$..."), or in close proximity ("...$w_1n_1n_2w_2w_3 \cdots w_n$..."). 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 of number of words found in a document, to the total number of words of a term, before the term is assigned to the document; we refer to it as the *partial match threshold*. 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 5 worked well for two-word phrases in english [9]. It has been our experience that indexing works best when both parameters depend on the size of the term.

This method was applied to the "Description" attribute using the vocabulary produced by the algorithm described in § 4.2.1. As it turned out, the quality of indexing for the "Description" attributes did not really matter because the attribute was used only twice (see section 4.1).

4.3. 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, component retrieval controlled experiments usually use queries that correspond closely to component specifications. This does not reflect normal usage for a reusable components library tool because. 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 [17]. 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

2. Computer science being a relatively new field, most concepts are still described by noun phrases, as in "Software Engineering" "Bubble Sort", "Printing Monitor", etc., rather than single words as is the case for more mature fields such as medicine; see [23] for a fascinating treatment of the evolution of languages and terminology.

application domain using textbook-like language[3].

4.4. Component relevance

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. Within the context of 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 [24], which depends, among other things, on the users' 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 irreplacable: the user needs all of them. In other words, assuming that a query Q has N relevant documents, and retrieved a set of documents $S = \{D_1,...,D_m\}$, we can define *pertinence* or *usefulness*, and recall as follows:

$$PERT(D_i) = \begin{cases} \frac{1}{N}, \text{ if } D_i \text{ is relevant} \\ 0, \text{ if } D_i \text{ is not relevant} \end{cases}, \text{ and}$$

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

With software component retrieval, the notion of 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 $S_1,...,S_k$, where $S_i = \{(D_{i_1},D_{i_2},...,D_{i_k}\}$. For each i=1,...,k, we have, as a first approximation:

$$PERT(S_i) = PERT(D_{i_1},D_{i_2},...,D_{i_{k_i}}) = \sum_{j=1}^{k_i} PERT(D_j/S_i) = 1 \quad (A)$$

where $PERT(D/S_i)$ 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 documents, 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. $PERT(D_i/S_j) = \frac{size(D_i)}{size(S_j)}$. In this case $PERT(D_1/S_1) = 0.6$, $PERT(D_2/S_1) = 0.4$, $PERT(D_1/S_2) = 0.3$, $PERT(D_3/S_2) = 0.4$, and $PERT(D_4/S_2) = 0.3$. Assume that a query retrieves the component D_1. In this case, $PERT(D_1) = Max(PERT(D_1/S_1),PERT(D_1/S_2)) = 0.6$. If the query retrieved D_1 and D_3, instead, $PERT(\{D_1,D_3\}) = Max$ $(PERT(\{D_1,D_3\}/S_1), PERT(\{D_1,D_3\}/S_2)) = Max\ (PERT(D_1/S_1) + PERT(D_3/S_1), PERT(D_1/S_2) + PERT(D_3/S_2)) = Max\ (0.6 + 0.0, 0.3 + 0.4) = 0.7$. 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 $S_1,...,S_k$, a query that returns a set of components S has the pertinence:

$$PERT(S) = \max_{j=1,...,k} PERT(S \cap S_j/S_j) \quad (B)$$

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 $PERT(D_1,D'_2) = 0.6 + 0.3 = 0.9$. 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 answered 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 PERT(.)) to the total number of retrieved documents. 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.

4.5. Performance results

Table 1 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, while making sure that 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 4 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.

Intuitively, 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 exceptions. 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 [1]. Next, we isolated the effect of the search type to see if the difference

3. The subjects were mostly french-speaking, while the library's documentation and controlled vocabularies were in English.

in recall and precision performance is significant. The results are shown in Table 2.

Q.	Full-text retrieval			Keyword retrieval		
	Subj	Rec	Prec	Subj	Rec	Prec
1	3	100	88.66	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	33.33	33.33	2	12.5	25
7	2	65	75	3	66.33	50
8	2	30	75	3	30	83.33
9	3	53.33	100	2	30	78
10	3	78.33	80.33	1	35	100
Avg	(26)	63.49	74.47	(23)	42.41	68.33

Table 1. Summary of retrieval results.

Effect of search method	Recall	Precision
F Value	4.1	0.93
Pr > F	0.0500	0.3404

Table 2. Significance of differences between plain-text retrieval and keyword retrieval.

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 [2, 24, 25], although the difference hardly justifies the extra costs involved in controlled vocabulary-based indexing and retrieval [25]. Similarly, a comparative retrieval experiment for reusable components conducted by Frakes and Pole at the SPC showed that recall values were comparable, and a superior precision for controlled vocabulary-based retrieval [6]. Most surprising in our results is the significant difference is recall performance.

We sought to explain the counter-intuitive/evidence difference in recall performance. We first note that out of the 11 queries, some were supposed to retrieve single components (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 indiscriminantly methods and classes. With controlled-vocabulary search, users have to specify the kind of components they are seeking (a class or a method), since the two types do not support the same set of attributes/facets; this makes the search more tedious and users may give up easily. For this explanation 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 3 compares the two kinds of queries.

Query set	Full-text		Keyword	
	Rec	Prec	Rec	Prec
1 comp. q.	62.08	84.67	44.08	65.333
many comp. q.	64.43	68.17	41.30	70.33

Table 3. Comparing the two sets 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 methods, since the same query would retrieve both kinds of components. Table 4 shows recall and precision values for the two methods, separated into the two kinds of queries.

Query set	Full-text		Keyword	
	Rec	Prec	Rec	Prec
Q. w. methods	41.333	59.17	27.77	47.27
Q. w. classes only	85.65	89.77	57.05	89.40

Table 4. Comparing the two sets of queries depending on whether they retrieve methods or not.

Interestingly, there is a marked difference in performance between the two groups of queries. However, in both cases, plain-text retrieval is markedly superior to controlled-vocabulary retrieval with regard to recall (and precision, for the case of queries retrieving methods). Another possible explanation for the lower performance of controlled-vocabulary based retrieval is related to the quality of indexing, but the results didn't seem consistent with the hypothesized effect on retrieval performance [19].

5. Discussion

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 as good* retrieval performance [6]. We attempted to bring the two kinds of methods to a level-playing field by: 1) automating as much of the pre-processing involved in controlled vocabulary-based methods as possible to address the costs issue, 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; see section 4.2), and 2)

3. Frakes and Pole compared 4 methods, and their test of statistical significance was based on variance analysis of the precision averages for the four methods, which was inconclusive [6]. 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 [6]), they would have established, statistically, the superiority of controlled-vocabulary retrieval.

the fully automatic free text search performed better than the fully manual controlled-vocabulary based indexing and retrieval of components.

Because this result is somewhat counter-intuitive, we continue to analyze the results, which suggest, in some cases, a misunderstanding of the semantics of multiple-attribute queries, or multi-valued attributes, leading to what seemed to be a number of aimless queries with no clear search strategy. It thus appeared that the two-hour tutorial was not sufficient and users could have used some further experience with the toolset[4]. However, whichever additional effort we can put into, either the construction of the vocabulary, or the indexing of components, or the training of users of the tool, will only add to the costs of controlled vocabulary multi-faceted classification and retrieval, and we are not guaranteed, by any means, to achieve better results than with plain text search. Perhaps more importantly, four subjects out of five preferred plain-text search.

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.

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 à la Recherche* (FCAR), and Québec's *Ministère 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) provided us with invaluable assistance in measuring and interpreting the results.

References

1. Estelle Ah-Ki, in *Reutilisation de Composantes Logicielles Orientees-Objet*, Department of Computer Science, University of Quebec at Montreal, Montreal, Canada, July 1996. 150 pages
2. David Blair and M E Maron, "An Evaluation of Retrieval Effectiveness for a Full-Text Document-Retrieval System," *Communications of the Association for Computing Machinery, 28, 3*, pp. 289-299, March 1985.
3. Doug Cutting, Julian Kupiec, Jan Pedersen, and Penelope Sibun, "A Practical Part-of-Speech Tagger," in *Proceedings of the Applied Natural Language Processing Conference*, 1992.
4. D Denning, J Minker, A Parker, A Ralston, E Reilly, A Rosenberg, C Walston, T Willoughby, J Sammet, and A Blum, "The Proposed New Computing Reviews Classification Scheme," *Communications of the Association of Computing Machinery*, vol. 24(7), pp. 419-434, July 1981.
5. Graham Dumpleton, in *OSE - C++ Library User Guide*, Dumpleton Software Consulting Pty Limited, Parramatta, 2124, New South Wales, Australia, 1994. 124 pages
6. William B. Frakes and Thomas Pole, "An Empirical Study of Representation Methods for Reusable Software Components," *IEEE Transactions on Software engineering*, pp. 1-23, August 1994.
7. Robert Godin and Hafedh Mili, "Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices," *ACM SIGPLAN Notices*, vol. 28, no. 10, pp. 394-410, Washington, D.C., 26 Sept - 1 Oct, 1993. OOPSLA'93 Proceedings
8. Robert J. Hall, "Generalized Behavior-based Retrieval," in *Proceedings of the 15th International Conference on Software Engineering*, pp. 371-380, ACM Press, Baltimore, Maryland, May 17-21, 1993.
9. Yoelle S. Maarek, Daniel M. Berry, and Gail E. Kaiser, "An Information Retrieval Approach for Automatically Constructing Software Libraries," *IEEE Transactions on Software Engineering*, vol. 17 (8), pp. 800-813, August 1991.
10. Ali Mili, Rym Mili, and Roland Mittermeir, "Storing and Retrieving Software Components: A Refinement-Based Approach," in *Proceedings of the Sixteenth International Conference on Software Engineering*, Sorrento, Italy, May 1994.
11. Hafedh Mili and Roy Rada, "Building a Knowledge Base for Information Retrieval," *Proceedings of the Third Annual Expert Systems in Government Conference*, pp. 12-18, October 22-25, 1987.
12. Hafedh Mili, in *Building and Maintaining Hierarchical Semantic Nets*, The George Washington University, August 1988. Doctoral Dissertation
13. Hafedh Mili and Manon Grenier, "Managing Documentation for Software Reuse," *Information and Decision Technologies*, vol. 18, pp. 115-134, 1992.
14. Hafedh Mili and Haitao Li, "Data Abstraction in SoftClass, an OO CASE Tool for Software Reuse," in *Proceedings of TOOLS'93*, ed. by Bertrand Meyer, pp. 133-149, Prentice-Hall, Santa-Barbara, CA, August 2-5, 1993.
15. Hafedh Mili, Roy Rada, Weigang Wang, Karl Strickland, Cornelia Boldyreff, Lene Olsen, Jan Witt, Jurgen Heger, Wolfgang Scherr, and Peter Elzer, "Practitioner and SoftClass: A Comparative Study of Two Software Reuse Research Projects," *Journal of Systems and Software*, vol. 27, May 1994.
16. Hafedh Mili, Odile Marcotte, and Anas Kabbaj, "Intelligent Component Retrieval for Software Reuse," in

4. The senior author of this paper, who did his Doctoral research on intelligent retrieval systems, still can't find a book using the University of Quebec's library bibliographic retrieval system, after 8 years on the faculty.

Proceedings of the Third Maghrebian Conference on Artificial Intelligence and Software Engineering, pp. 101-114, Rabat, Morocco, April 11-14, 1994.

17. Hafedh Mili, Fatma Mili, and Ali Mili, "Reusing Software: Issues and Research Directions," *IEEE Transactions on Software Engineering*, vol. 21, no. 6, pp. 528-562, June 1995.
18. Hafedh Mili, William Harrison, and Harold Ossher, "Supporting Subject-Oriented Programming in Smalltalk," in *Proceedings of TOOLS USA '96*, Santa-Barbara, CA, July 29 - August 2, 1996.
19. Hafedh Mili, Estelle Ah-Ki, Robert Godin, and Hamid Mcheick, "Representing and retrieving reusable components," *IEEE Transactions on Knowledge and Data engineering*, October 1996. Submitted (revised version of 1994 draft).
20. Hafedh Mili, Houari Sahraoui, and Ilham Benyahia, "Representing and Querying Object Frameworks," Technical report, Dept of Computer Science, University of Quebec at Montreal, May 1996.
21. Ruben Prieto-Diaz and Peter Freeman, "Classifying Software for Reusability," *IEEE Software*, pp. 6-16, January 1987.
22. Roy Rada, Hafedh Mili, Ellen Bicknell, and Maria Blettner, "Development and Application of a Metric on Semantic Nets," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 19(1), pp. 17-30, January/February 1989.
23. James Roskind, *The C++ Grammar*, July 1991.
24. Gerard Salton and Michael McGill, *Introduction to Modern Information Retrieval*, McGraw-Hill, New York, 1983.
25. Gerard Salton, "Another Look at Automatic Text-Retrieval Systems," *Communications of the Association of Computing Machinery*, vol. 29(7), pp. 648-656, July 1986.
26. Scott N. Woodfield, David W. Embley, and Del T. Scott, "Can Programmers Reuse Software," *IEEE Software*, pp. 52-59, July 1987.
27. Amy Moormann Zaremski and Jeannette M. Wing, "Signature Matching: A Key to Reuse," *Software Engineering Notes*, vol. 18, no. 5, pp. 182-190, 1993. First ACM SIGSOFT Symposium on the Foundations of Software Engineering
28. Amy Moormann Zaremski and Jeannette M. Wing, "Specification Matching: A Key to Reuse," *Software Engineering Notes*, vol. 21, no. 5, 1995. Third ACM SIGSOFT Symposium on the Foundations of Software Engineering

Object Oriented Reuse
&
Reuse on the Internet