A Hierarchy-Aware Approach to Faceted Classification of Object-Oriented Components

E. DAMIANI
Università Statale di Milano—Polo di Crema
M. G. FUGINI
Politecnico di Milano

C. BELLETTINI
Università Statale di Milano

This article presents a hierarchy-aware classification schema for object-oriented code, where software components are classified according to their *behavioral characteristics*, such as provided services, employed algorithms, and needed data. In the case of reusable application frameworks, these characteristics are constructed from their *model*, i.e., from the description of the abstract classes specifying both the framework structure and purpose. In conventional object libraries, the characteristics are extracted semiautomatically from class interfaces. Characteristics are term pairs, weighted to represent "how well" they describe component behavior. The set of characteristics associated with a given component forms its *software descriptor*. A descriptor base is presented where descriptors are organized on the basis of structured relationships, such as similarity and composition. The classification is supported by a thesaurus acting as a language-independent unified lexicon. The descriptor base is conceived for developers who, besides conventionally browsing the descriptors hierarchy, can query the system, specifying a set of desired functionalities and getting a ranked set of adaptable candidates. User feedback is taken into account in order to progressively ameliorate the quality of the descriptors according to the views of the user community. Feedback is made dependent of the user typology through a *user profile*. Experimental results in terms of recall and precision of the retrieval mechanism against a sample code base are reported.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

This work was partially supported by the Italian National Research Council in the framework of "Progetto Coordinato Ambienti di Supporto alla Progettazione di Sistemi Informativi," 1995–1998.

Authors' addresses: E. Damiani, Università Statale di Milano—Polo di Crema, Via Bramante 65, Crema (CR), Italy; email: edamiani@crema.unimi.it; M. G. Fugini, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Via Ponzio 34/5, Milano, 20133, Italy; email: fugini@elet.polimi.it; C. Bellettini, Dipartimento di Scienze dell'Informazione, Università Statale di Milano, Via Comelico 39, Milano, 20100, Italy; email: belletc@dsi.unimi.it.

General Terms: Documentation

Additional Key Words and Phrases: Code analysis, component repositories, component retrieval, software reuse, user feedback

1. INTRODUCTION

Research about classification and retrieval techniques for software components has undergone a notable development in recent years. The original view of software reuse and commercial interest in object-oriented technology were centered on reuse of single components [Cox 1986]. In this view, the critical issues are the identification of the appropriate granularity of the reusable component [Biggerstaff and Perlis 1989] and the understanding and adaptation of reuse components to new application requirements. Meanwhile, some organizations have already based their development process on large internally developed software libraries (for example, the Bell Northern Research library [Cashin 1991] and the Asset Aerospace library [Lillie 1991]). Nowadays, various projects and commercial systems exist, supporting the concept of huge reusable components malls; a review of techniques for library management is presented in Mili et al. [1998]. Today, among the most promising approaches toward reuse, the paradigm of *application frameworks* is gaining wide acceptance in the object-oriented community [Johnson 1997]. Application frameworks are application skeletons that can be customized by the application developer. They can still be considered as components, in that they are sold as products and that an application can use various frameworks. However, they are a more powerful concept which can spare the effort of composing and adapting heterogeneous components in order to build a new application. In fact, they provide mechanisms such as custom composition of selected subclasses and implementation of methods which either plug into, or override, methods in the superclasses. Moreover, they provide a reusable domain for finer-granularity components because, within a framework, each component makes assumptions about its environment and can hence be readily used together with other components.

Currently both styles of reuse, that is, traditional, library-based reuse and framework-based reuse, exist jointly in real development environments, and borders are often blurred. Therefore, in this article we address both styles and make explicit distinctions when appropriate. Whatever the reuse technique, the problem arises of locating and adapting the useful candidates on the basis of some specification of their behavior [Krueger 1992; D'Souza and Wills 1997; Mili et al. 1997]. These problems have recently assumed relevance in projects requiring access to large-scale systems of components and frameworks, as reported for instance in Bäumer et al. [1996].

Classification and selection are considered as a key success factor for software reuse projects, especially when reuse involves also software arti-

facts, besides code [Basili and Rombach 1991; Prieto-Díaz 1993; Pfleeger 1996; Damiani et al. 1997]. Moreover, we believe that correct component classification can help to address several other problems, besides reuse, such as code comprehension for reverse engineering, dynamic domain modeling, evaluation of programming language dependencies, and usage patterns. It is the purpose of this article to propose two consistent models, respectively, for classification and retrieval of class libraries and application frameworks stored in an object-oriented code base. These models have been conceived based on the following six principles:

(1) descriptor-based behavioral classification

(2) controlled granularity,

(3) language independence,

(4) trainable user-adaptive response,

(5) support for both query and navigational interfaces, and

(6) thesaurus-based controlled vocabulary.

Both models are based on *software descriptors* (SDs) which have an attribute part describing usual *nonbehavioral* characteristics of code (for instance, author, programming language, execution environment, and performance data). SDs contain also a term-list-based part, representing *behavioral* properties, such as provided services, employed algorithms, and needed data. This article focuses on this second portion of SDs, which play much the same role as sets of *open class keywords* [Maarek et al. 1991] or *type models* [D'Souza 1996] describing the component behavior. Moreover, they are indeed structured, since their elements are not single keywords but *weighted* pairs of related terms (*features*). Weights are the mechanism allowing for highlighting the characteristics that are particularly relevant to the specification of the component behavior with respect to more generic or less important ones. The weights system makes it possible to pose queries where the user asks for a set of characteristics expected from the component. To each feature, the user associates a weight expressing the importance of its presence in the retrieved components.

In the classification model, in order to build SDs features, we rely on semiautomatic extraction of terms from code. SDs are manually partitioned by the system maintainers, called *application engineers*, according to preset *contexts* (or facets or categories [Prieto-Díaz and Freeman 1987]), thus supporting a language-independent, *faceted* classification technique.

In the two reuse models (component based and framework based), SDs are associated by the extraction procedure respectively to those components that are considered by the application engineer as reuse units, or to the framework hierarchy as a whole. An initial setting for weights values is provided by a *feature weighting function* (FWF) adapted from the one proposed in Salton and Buckley [1988]. Weights, and the whole system of descriptors, attain their correct values along the system life cycle by

observing the user reactions to query answers. This mechanism ensures user-adaptive response, at least after an adequate training phase.

In the retrieval model, we exploit partial matchings between the set of weights associated to the query and those associated to available components. A *similarity* index is defined as the level of matching between the query, and candidates are returned according to their similarity to the searched component(s). This takes into account the fact that, while the "perfect" candidate seldom exists in the code base, components are often available which provide the desired functionalities "to some extent." We use the term *compatibility* (rather than similarity) due to asymmetry in the object model: the functionalities of children classes are also the functionalities provided by their ancestor(s) in the Is-a hierarchy, while the reverse usually does not hold.

The whole lexicon of terms used in the SDs is stored in a thesaurus. The thesaurus *semiautomatic initialization* consists of computing the relevance of each term in all contexts using a *context relevance function* (CRF)and exploiting the results to evaluate *synonymy* between pairs of terms. This allows for automatic query expansion based on synonymy, avoiding in many cases unwanted system silence; but the thesaurus also proved useful as a browsable controlled vocabulary, putting the user in touch with the system lexicon.

The article is organized as follows. In Section 2, the faceted construction of hierarchical SDs is described together with the weighting procedure for features. In Section 3, the notion of compatibility between SDs is introduced as the basis for component analysis. In Section 4, procedures for the automatic initialization of the thesaurus and for its semiautomatic filtering are described. In Section 5, a method for adaptive tuning of the approach according to user feedback is presented, and the semantics used for query formulation is briefly discussed. In Section 6, a suite of test queries and their results in terms of recall and precision parameters are examined. In Section 7 a comparison of our approach with other related approaches in the fields of software engineering, information systems, and information retrieval is given. Finally, in Section 8 some concluding remarks, implementations issues, and observation on future work are given. In the Appendix, a sample query block used for retrieval validation and an evaluation of the thesaurus are presented.

2. SOFTWARE DESCRIPTORS

The role of software descriptors in our approach is to give a synthetic view of the behavioral properties of components. SDs are constructed from components and other available documentation. By *component* we mean either a class or class cluster or an application framework which constitute a *classification* and *retrieval unit*. An SD is a list of terms pairs, called *features*, describing the component behavior. Each feature in the list has an associated *weight* computed using a relevance assignment technique adapted from information retrieval [Salton 1989]. The two-term structure

for features allows for a richer semantics than the usual single-term description [Salton 1989], while remaining simple enough to avoid the need for a formal grammar and semantics definition of a component description language. It is worth remarking, that, from the lexical point of view, our feature list is equivalent to more complete documents extracted from code by means of natural-language parsers [Etzkorn and Davis 1997] and therefore can be processed using techniques adapted from information retrieval methods. In fact, automatic construction of documents from code relies on some well-known nomenclature conventions used in OO programming environments [Booch 1994], which ensure that in the object interfaces there is a high percentage of active verb phrases, i.e., of verbs, nouns, and adjectives. While those approaches [Biggerstaff et al. 1994; Etzkorn and Davis 1997] aim at reconstructing a document structure from code by adding connectives, such as articles and prepositions, our weighting technique operates satisfactorily also on SDs only. In fact, connectives would be filtered out by any relevance computation technique.

Features in an SD are *unique pairs*. Consistency in the initial choice of features is the responsibility of application engineers; some kind of thesaurus-aware editor may help to enforce it. Note that uniqueness of pairs does not require or imply uniqueness in term usage, for instance, *administer-system* is a different (and perfectly admissible) feature than *administer-network*.

As far as classification granularity is concerned, i.e., the identification of the code unit that deserves an SD, we observe that there are some basic differences between the framework style and the more traditional, library-based style of reuse. Indeed, traditional OOP reuse focuses on factoring out common parts of behavior into superclasses, while implementing concrete behavior in the derived classes. So, the behavior of the component is specified for each component, at all levels of the hierarchy, and hence SDs are attached to single classes or optionally to class clusters (computed with methods analogous to those presented in Batory and O'Malley [1992], Batini et al. [1993], Bellinzona et al. [1995], and Castano and De Antonellis [1993]). In the framework style of reuse, the technique of the plug-in methods is exploited as follows. In the top classes, i.e., in the more abstract classes of the framework, methods are reported whose implementation is deferred to lower levels of the hierarchy. Hence, the top classes contain all the information necessary to provide an SD of the whole framework.

SDs may span over several *contexts*, each represented by a label, thus supporting a *faceted classification* and *retrieval* technique. Contexts are proposed in this article as a *policy-free* tool for the application engineer to divide the descriptor base in a preset number of partitions; however, their number can be increased according to later needs. While contexts are considered as a flat list of labels when identifying the lexical properties of the descriptor base, they can be used to support a variety of hierarchical faceted or context-based techniques in the classification process. An application of contexts to the faceted classification method is illustrated in the following example. The descriptor base of a general-purpose set of low-level

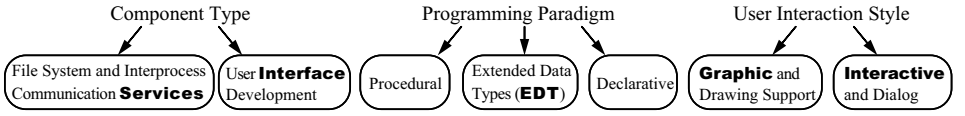


Fig. 1. Selection of contexts from a faceted classification.

components for Microsoft Windows applications development can be partitioned according to the classification forest of facets shown in Figure 1.

After adopting such a classification forest, the application engineer specifies the values that label the leaves of each tree in the classification, obtaining a flat list of context values to be used for the lexical analysis of the descriptor base. Often, a component will be classified under several contexts; in fact, these contexts are refinements of *views* on the descriptor base [Prieto-Díaz and Freeman 1987], as it is usually the case for faceted classification. Moreover, the context list itself is extendible: if a bunch of components for, say, network computing are to be added to the descriptor base at a later stage, the new context value is added as a leaf to the component type classification tree, and a **Network** entry is added to the context list. Contexts can also be used to exploit for classification purposes the concept of *role* used in some reuse paradigms [Bäumer et al. 1997]. In fact, a component can be classified under all the contexts where it can play a role without using the role itself, which is sometimes too fine grained, as a classification category. For example, in a descriptor base partitioned in two values of the functional areas [Prieto-Díaz and Freeman 1987] facets `Banking` and `Insurance`, a `Client` component endowed with three roles "Investor," Borrower," and "Policy-Holder" will be classified in both contexts, since the first two roles hint to the `Banking` context.

While maintaining the descriptors, application engineers view SDs as lists of features describing component functionalities; while looking for components, developers use SDs as requirement lists. In the former case, SDs are *statically associated to components*, while in the latter case they are *dynamically associated to queries*.

The flow of information in the classification system is depicted in Figure 2.

An extraction protocol examines the object code libraries and the thesaurus storing the available terms and context names. The protocol produces SDs, which can be faceted and tuned by the application engineer. SDs are then stored in a descriptor base. In the figure, the process of thesaurus construction is also depicted: the extraction protocol extracts lists of terms from the object libraries, and a suitable function (context relevance function (CRF)) computes the relevance of terms in the various contexts. The CRF values are used to evaluate synonymy between pairs of terms producing an initialization of the thesaurus which can be later tuned by the application engineer.

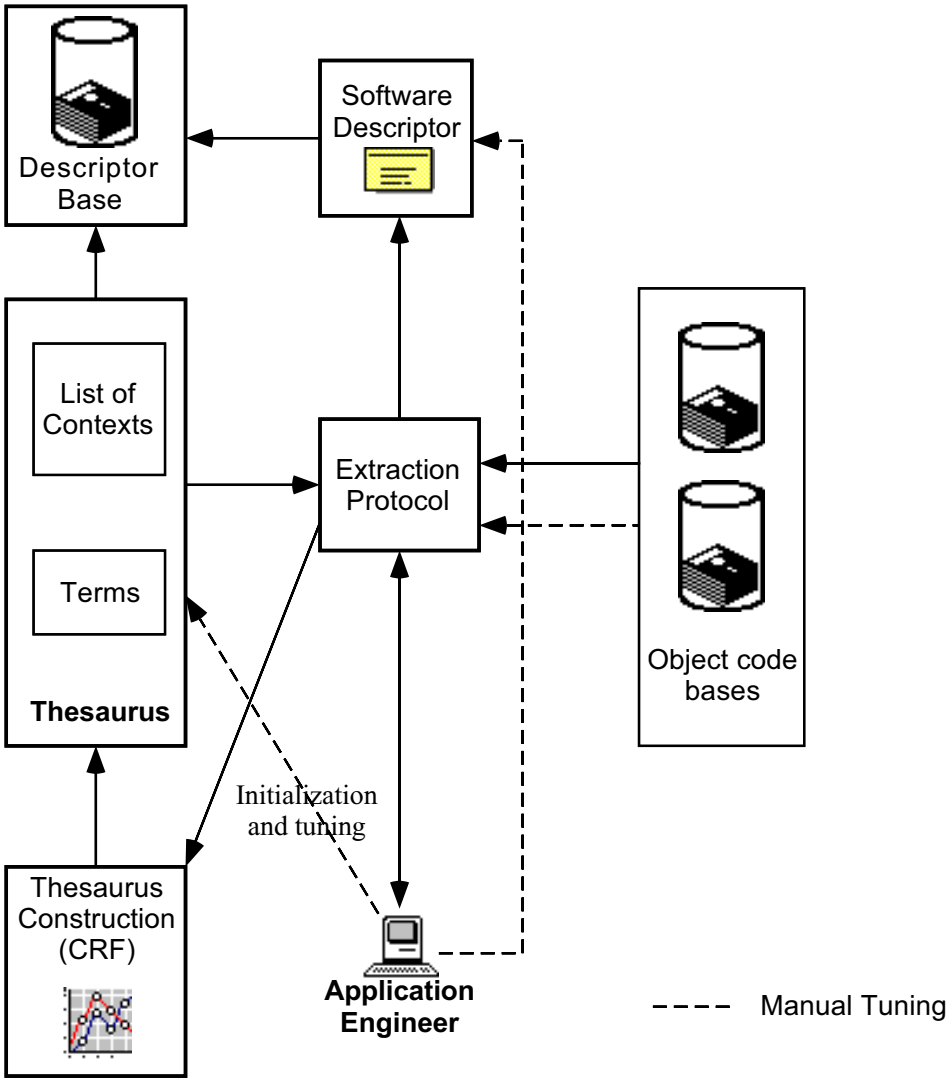


Fig. 2. Information flow in the classification system.

2.1 Features

A descriptor SD associated to a software component or framework in the descriptor base has the following format:

$$\mathbf{SD} = \mathbf{C_1}\ \texttt{list of [feature : weight]}$$

$$\mathbf{C_2}\ \texttt{list of [feature : weight]}$$

$$\ldots$$

$$\mathbf{C_C}\ \texttt{list of [feature : weight]}$$

where (possibly empty) lists of features are not necessarily disjoint and where `feature = term1,term2`. Terms in the feature are ordered. Lists of features in a context contain no duplicated features. The fact that a feature can appear repeatedly in an SD is due to the fact that SDs are multicontext; as we will see, features can be attributed to several contexts.

Features are provided as a policy-free mechanism, and their use is bound to the scope where the description is formulated. A possible use of the term pairs consists in providing a signature-like behavioral specification, with the advantage of a good degree of readability for the user. In fact, feature terms can be read as a "`Verb,Noun`" pair, where the `Verb` field describes the type of functionality, or service, of the component and the `Noun` field describes the type of component upon which the functionality is performed. Examples of `Verbs` are `edit`, `set`, `create`, and `initialize`. Verbs can also describe procedures of applications, such as `handle letter` and `check account`. Examples of `Nouns` are `bytestream`, `state`, `account`, and `document`.

Although in the remainder of this article we will not elaborate on the usage of term pairs, we remark that these categories of terms can be used to organize the thesaurus in partitions to be searched separately allowing synonymy to be computed only between terms belonging to the same category. This partition can also facilitate the formulation of queries by means of a thesaurus-aware editor: when a `Verb` is entered as the first term of a query feature, the system suggests the list of `Nouns` that are associated to this `Verb` in the descriptor base.

Another possible use is to include nonfunctional properties of a component in its SD; in this case, a feature can be read as a "`Noun,Adjective`" pair. Examples are the features "`maturity,high`" or "`documentation, good`" describing two nonfunctional properties of components. This kind of feature may be obtained from documentation or, more frequently, is added manually by the application engineer.

2.2 Features Extraction

The list of features composing the SD of each component is automatically extracted from code and tailored by the application engineer (see Figure 2). The extraction is performed with the assistance of a thesaurus-aware editor. Here we describe an inheritance-aware protocol aimed at extracting features from object-oriented code in both traditional and framework-based reuse. Following Batory and O'Malley [1992] and Spanoudakis and Constantopoulos [1994], among others, our classification schema follows a *hierarchical approach*.

2.2.1 *Extraction Protocol*.

—The descriptor SD_i for the ith component is constructed by deriving a feature from each method. The *method name* is taken as the first term `term1` in the feature, and the first argument is taken as the second term

`term2` (destructors are excluded). If a method takes no argument, the second term of the feature takes the "null" value (represented by −).

—The Is-a hierarchies are used to mirror components hierarchies into a *hierarchical organization* of descriptors [Batory and O'Malley 1992]. As the properties of a component are represented by the entire path in its Is-a hierarchy, analogously its description can be traced back by following the entire chain of descriptors in its classification hierarchy. Hence, SDs are linked via *Is-a connections*: whenever a component is linked to others in Is-a, these links are mapped into Is-a links of the corresponding SDs.

—To deal with *framework style reuse*, calls to deferred methods are also examined. An SD is obtained taking the name of the deferred method and its first parameter. Regarding composition information, the SDs of a component also include the SDs of its member classes.

—Other features of the SD can be extracted manually from the header of a class and textual comments.

—Method names like `store`, `print`, which are found in almost every class, are called *generic terms* or *stop-words*. These terms are filtered by the weighting function, which assigns them a low weight.

For library-style reuse, the Is-a hierarchy of the object library is used to produce the hierarchy of descriptors. In particular, the Is-a links existing in the library are mirrored in Is-a links of the descriptor base among the descriptors of the classified components. Hence, the hierarchy of SDs is a subgraph of the original inheritance graph of the library. Regarding the effect of inheritance upon the contents of SDs, the default is that features are propagated to subclasses, so that each SD contains all the features of its ancestor SDs although these are not explicitly shown. So, during retrieval, the highest component in the hierarchy exhibiting the desired features is returned. This is due to a mechanism of weights lowering along Is-a hierarchies (see Section 2) that progressively decreases the weights of inherited features from the ancestor into the SD of the descendant.

Now, we show some sample SDs constructed by examining the components belonging to the standard *Visual C++ Foundation Class Library*. The first SD is shown in Figure 3. It describes **OLE Document**, a versatile component of the Microsoft Class library that can be used either as a compound document implementation, i.e., a software communication device between applications according to the OLE-DCOM standard, or as a container for document items as a building block for specialized document-processing applications. This SD is a typical example of descriptors for library-based reuse, where most of the significant behavioral specifications appear in the concrete classes, like the OLE Document one, at the lower levels of the hierarchy while the top classes factor out only a limited amount of behaviors. This component is also an example of repeated features in different contexts. In fact, the component is classified under

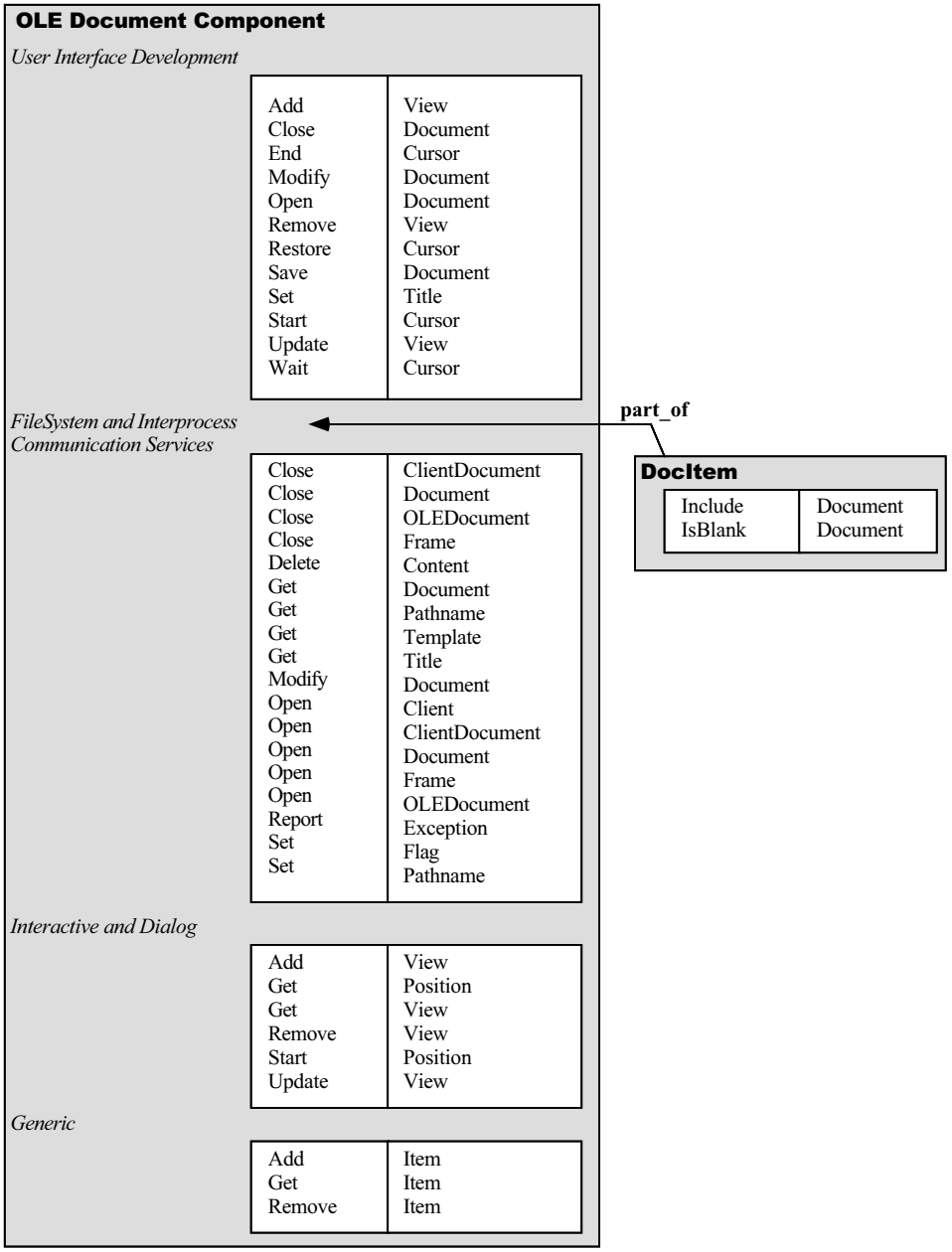


Fig. 3. SD of the OLE Document component with features classified in four contexts. The SD of the DocItem component is also shown; its features contribute to enrich the OLE Document SD.

four contexts: `User Interface Development`, `FileSystem and Interprocess Communication Services`, `Interactive and Dialog`, and `Generic`. The first two contexts are values of the component type facet mentioned in the previous sections (see Figure 1); `Interactive and`

`Dialog` is a value of the user interaction style facet. The `Generic` context copes with general-purpose features that carry limited behavioral information, but can still be useful for browsing the descriptor base, as we will see in the section devoted to retrieval.

Both `User Interface Development` and `FileSystem` and `Interprocess Communication Services` contexts are inherited from the ancestor document class which is a key component of the Microsoft Class library for development of graphical applications with a windowing interface. Since OLE Document provides some high-level user-interaction capabilities, these have been classified under the `Interactive` and `Dialog` context.

The descriptor includes several multicontext features such as `Modify, Document` and `Open,Document` which specify behavioral characteristics that can be useful both in user interface development and as file-system-related services in document-processing applications. Other multicontext features are `Add,View` and `Update,View` in the `Interactive` and `Dialog` and `User Interface Development` contexts. Moreover, in this SD the extraction method includes the features of member classes like **DocItem**, that is, `Include,Document` and `IsBlank,Document`. This contributes to make the SD a characterization of the component behavior which is richer than the interface alone. The length of the SD, which amounts to about 80 terms, is a rather typical value for this kind of component.

Coming to framework style reuse [D'Souza 1996], where deferred methods are included in the top-most level classes, we notice that it provides even lexically richer SDs.

As an example, in Figure 4, we show a simplified yet lexically rich SD constructed from **TControl**, an abstract class of a well-known application framework (the Virtual Component Library defined by Borland's Delphi development environment [Cantù 1996]). **TControl** is the root of a class hierarchy comprising `TWinControl`, used by Delphi programmers as the basic building block for Windows application development (and the component has no part-of links). As one could expect when dealing with framework style reuse, we observed that these features are shared by nearly all the control components belonging to the framework. The descriptor was extracted both from **TControl**'s published *properties* and *methods*. For the sake of simplicity, this component belongs to a single context (`User Interface Development`).

2.3 Contexts Propagation

Manual tuning of faceted SDs can be a tedious and time-consuming task for the application engineer. This burden can be partially relieved by the following interactive procedure which, under an initial choice of contexts performed by the application engineer, performs automatic propagation of contexts in the SDs following the Is-a hierarchy of the descriptors. This must be regarded as a minimal, although sometimes sufficient, context assignment technique. The result can then be manually enriched by the application engineer.

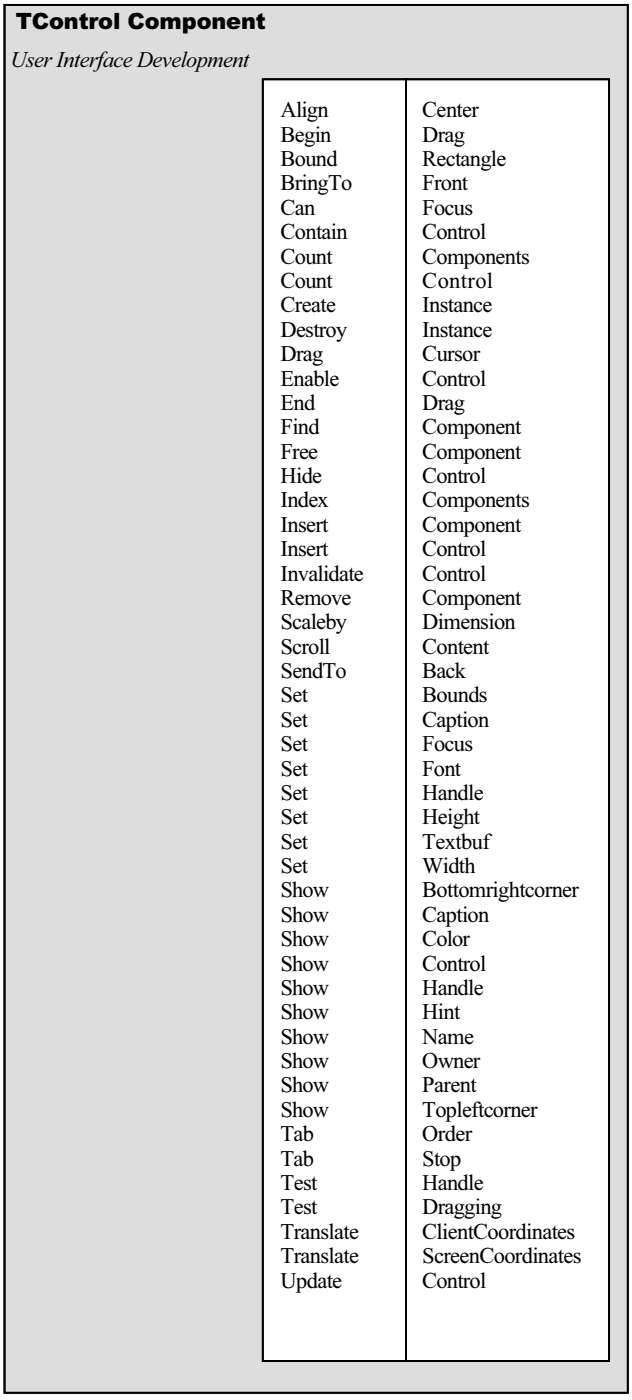


Fig. 4. SD of the **TControl** component belonging to the "Virtual Component" library. Only one context is shown for this SD.

This procedure puts into relation the Is-a hierarchy of a library with the classification tree used to select contexts. It works by considering that in the hierarchy of components, a descendant has partially the same behavior of its ancestors, as defined by inherited, nonmasked features. For example, a graphical editor component behaves as an editor, and therefore its inherited behavior is classified under the "text management" context (which is, for instance, a value of the component type facet). However, normally, a descendant exhibits its own behavior by locally defining new or masked methods. This new behavior can bring the component to belong to a new context. Most of the new methods of the graphical editor deal with graphical functionalities, making it belong to the `Graphic and Drawing Support` context as well. Accordingly, our propagation procedure automatically classifies inherited features under the same contexts they had in the ancestors, while leaving the application engineer free to assign contexts to the new behavior. This procedure is particularly suited for traditional style reuse where SDs have to be attributed to single components of a hierarchy. For framework style reuse, the procedure is often not necessary, since its coarser granularity allows for manual context attribution based on the top class descriptors.

As a first step, a set C of contexts is initialized for the descriptor base by the application engineer, following one of the approaches regarding the use of labels presented in the previous sections.

To further clarify how this context propagation works, we give an example (see Figure 5).

Suppose a geometric library is classified under the context $C_{init} = MATH$. The root of the library hierarchy is a `Coordinate_System` component with a "`compute,distance`" feature. At the first level of the Is-a hierarchy, a component managing `Cartesian_Coordinates` is included and since it performs matrix manipulation, the application engineer decides to assign it to the new context *LINEARMATH*. However, since the "`compute,distance`" feature is inherited (not masked) from the ancestor which is in the *MATH* context, "`compute,distance`" is kept in the *MATH* context as well. Instead, in the `Polar_Coordinates` component, the "`compute,distance`" feature is masked (i.e., the corresponding method is redefined in the component); consequently the feature is included in the *LINEARMATH* context, since its implementation is local to the `Polar_Coordinates` component. Suppose now that the descendant of the `Cartesian_Coordinates` component is `Cartesian_Axes`. Since this component draws the axes on the screen, the application engineer assigns it to the *GRAPHICS* context. The "`compute,distance`" feature is inherited, and *MATH* is propagated along the Is-a hierarchy. On the other hand, consider the descendant of `Polar_Coordinates`, i.e., `Polar_Axes`. Since it redefines the method associated to "`compute,distance`", the "`compute, distance`" feature is inserted in the SD of `Polar_Axes` under the *GRAPHICS* context, together with the other locally defined features. In

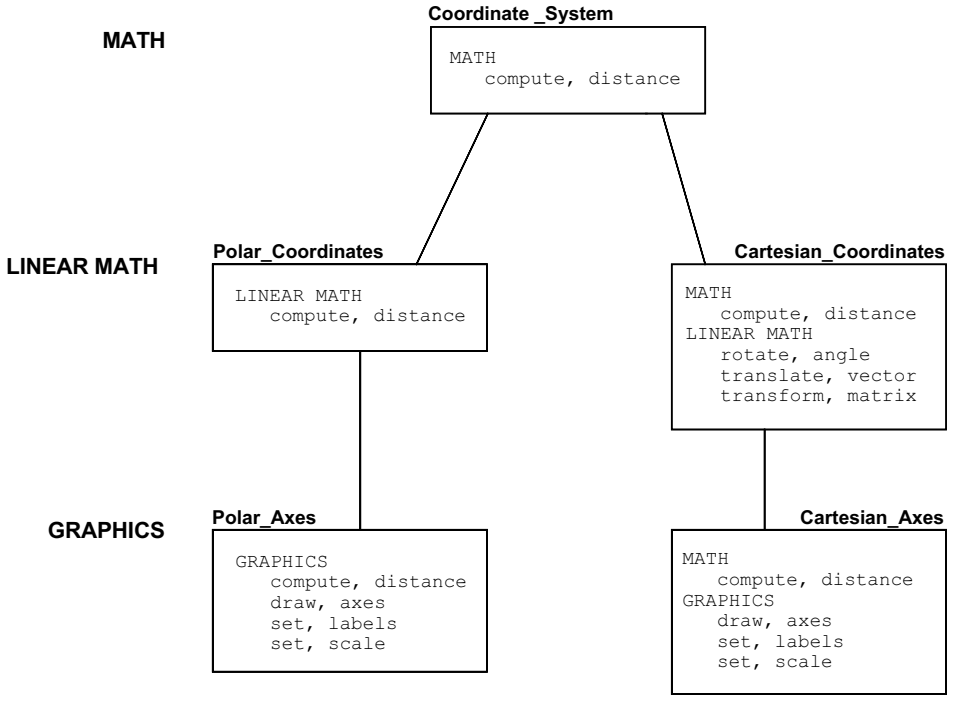


Fig. 5. Controlled context propagation for a geometric library: components are progressively classified in the MATH, LINEARMATH, and GRAPHICS contexts. Bold lines represents the `Is-a` hierarchy.

other terms, the context propagation procedure supports the classification of components under multiple contexts.

Going back to our example taken from the Microsoft library, consider context inheritance for the ancestor component of **OLE Document** (whose SD is shown in Figure 3), that is, **Document**, whose SD is shown in Figure 6. Moving along the Is-a hierarchy, inherited nonmasked features are repeated in the same contexts. New or masked features are classified in a suitable context of the classification tree. In traditional reuse, new facet values are usually necessary when descending along the Is-a hierarchy; in our example, this occurs for the `Interactive and Dialog` context. In framework-based reuse, instead, the complete set of facet values is specified for the top classes already.

2.4 Fuzzy Weights

The assignment of weights to features in the descriptor SD_i of the ith component is done automatically by the following algorithm.

2.4.1 *Weight Assignment Algorithm.*

(1) First, a weighting function is used in order to associate a weight $w_{i,k}$ to each feature f_k in SD_i. A suitable function is the following *feature weighting function* (FWF):

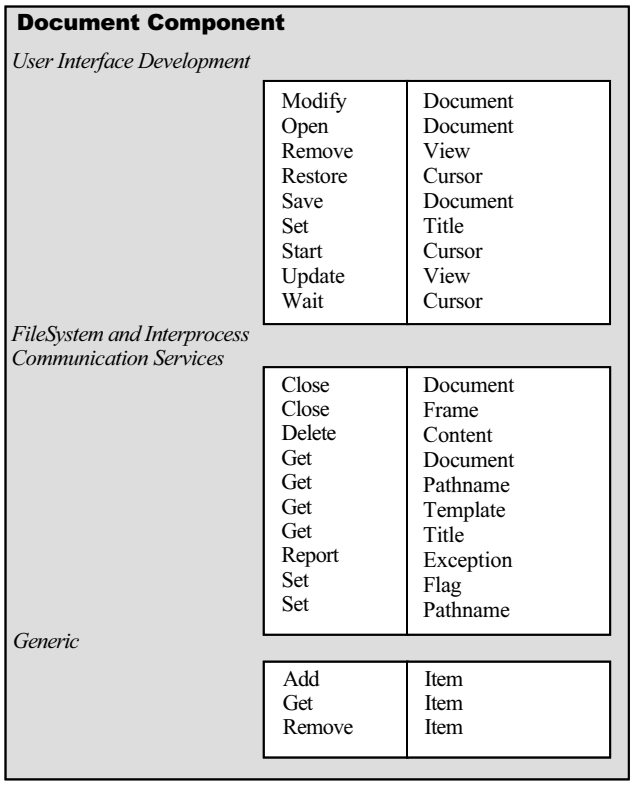


Fig. 6. SD of the document component.

$$w_{i,k} = \frac{\nu_{i,k} \log\left(\frac{N}{n_k}\right)}{\sqrt{\sum_{z=1}^{F}\left(\nu_{i,z} \log\left(\frac{N}{n_z}\right)\right)^2}}$$

which is the classical weighting function [Salton and Buckley 1988] used for relevance computation in document bases. In FWF, $w_{i,k}$ is the weight of the kth feature with respect to the ith component, and $\nu_{i,k}$ is the frequency of the kth feature in that component; N is total number of components, and n_k is the number of components exhibiting that feature. F is the total number of distinct features in the component base. The summation in the denominator, taken over all features of the descriptor base, is used as in Salton et al. [1994] for length normalization over SDs to ensure that all components have equal chance of being retrieved.

Schematically, FWF operates as follows: it assigns a weight $w_{i,k}$ to a feature f_k which depends on the ratio between the number of SDs where f_k occurs and the total number of occurrences of f_k in the whole

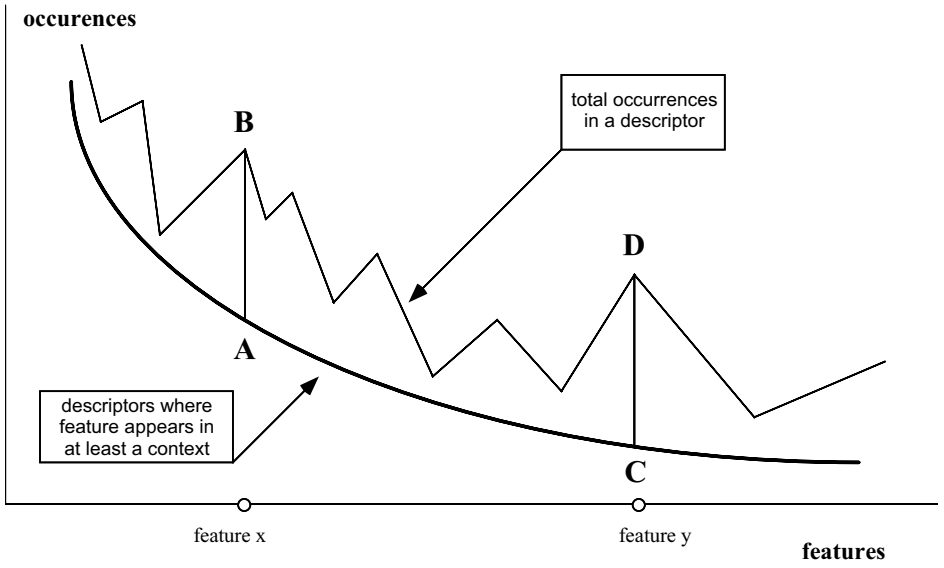


Fig. 7. Behavior of FWF.

descriptor base. These values are different because f_k can be repeated in the same descriptor in different contexts.

—*Is-a Hierarchies:* Along *Is-a* hierarchies, features are inherited with their contexts from the ancestor SD to the descendant SD. The weight of an inherited feature decreases of $1/2^h$, where h is the distance between the ancestor and the descendant on the inheritance graph.[1] Such a mechanism does not apply when features are redefined in the descendant; it is aimed at retrieving only the highest components in the hierarchy exhibiting a required feature f_k, rather than the entire chain of descendants where f_k is inherited.

The purpose of FWF is to *highlight*, within a descriptor, the features which are more relevant, i.e., more descriptive of the component behavior. Highlighting is obtained by taking into account the sum of the weights of all the features in each SD_i and increasing the weights of features belonging to descriptors having a low total weight.

Consequently, $w_{i,k}$ turns out to be higher when the difference between the total number of occurrences and the number of components exhibiting a feature f_k is obtained for a lower number of SDs. For example, Figure 7 represents the situation where segments AB and CD have the same length, i.e., the same difference between occurrences and number of descriptors, but feature x will be assigned less weight than feature y.

[1]In case of multiple inheritance, the selected value of h is the shortest inheritance path.

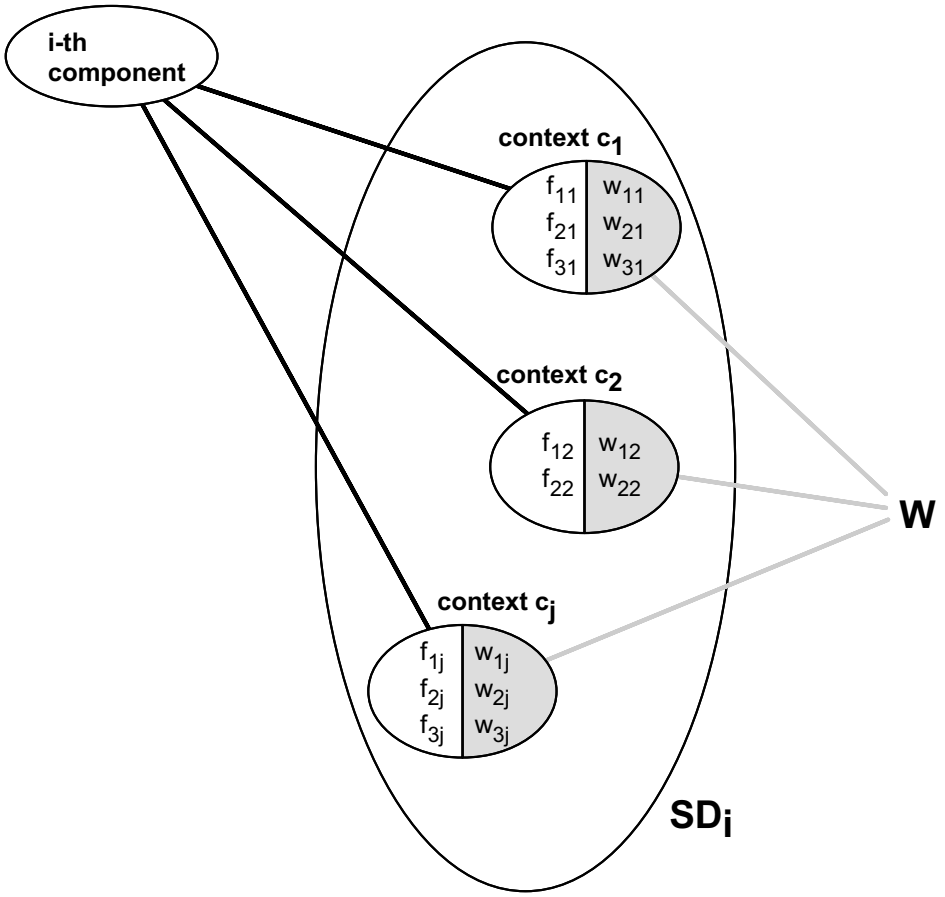


Fig. 8. Family W of fuzzy sets associated to contexts in the descriptor SD_i of the ith component.

(2) Second, exploiting *contexts*, we compute a *refinement* of FWF, namely $w_{i,k,j}$ by considering the products of $w_{i,k}$ times n_j/N, for $j = 1, 2, \ldots, C$, where n_j is the number of components in the jth context exhibiting the kth feature; N is the total number of components, and C is the number of contexts. We call this parameter n_j/N *context importance*. Each $w_{i,k,j}$ is the weight of the kth feature with respect to the ith component in the jth context.

Of course, some contexts might be empty in SD_i. Moreover, the case of SDs with no context can be dealt with simply by introducing a "generic" context.

(3) The sets of all the weights in the descriptor SD_i of the ith component is now interpreted as a family W of fuzzy sets $W_1, W_2, \ldots, W_C$. The family W is depicted in Figure 8.

The family W has characteristic functions $w_{i,k,j}$, $j = 1, 2, \ldots, C$. When using the fuzzy weights for component analysis, the union offamily members $\cup_j W_j$ will be computed corresponding to the contexts specified by the user. If no context is specified, the "generic" context is assumed.

In order to use fuzzy weights for our analysis, the set W_j corresponding to the selected context must be later defuzzified by extracting a value by means of a suitable function. Defuzzification will be described in Section 3.

3. COMPATIBILITY

Compatibility between SDs is an (asymmetrical) fuzzy relation called *confidence value*:

$$CV : SD \times SD \rightarrow [0,1]$$

It may well be seen as a function between components, in the sense of Yu [1975] or an extension of the boolean component match predicate defined in Moormann Zaremski and Wing [1997]. Fuzzy matching is one of the classical methods used in computing conceptual distance between objects descriptions (see for instance Bouchon-Meunier et al. [1996]). The main difference between our fuzzy compatibility and a classical matching function is *asymmetry*. Given two SDs R and D, in general we have that $CV(D, R) \neq CV(R, D)$. Asymmetry of the compatibility relation mirrors the fact that inheritance among components itself is asymmetrical: an instance of a parent class can always be replaced by an instance of a derived one, but not vice-versa. In fact, a child class exhibits all the features of the parent class but not vice-versa. Consequently, in a repository of object-oriented components, while a specialized item must be retrievable in response to a query for a more general one, a general component must not be retrieved when the descriptor base is queried for a specialized one.

Asymmetry influences many formal properties of compatibility, which closely resemble those of Is-a hierarchies; namely, it is a transitive, reflexive, and partial-order relation.

By this construction, values of importance in an SD can be seen as fuzzy logic (FL) values. FL (see for instance Klir and Folger [1988] and Kosko [1992]) is equipped with standard operators (AND, OR, $\Rightarrow$, $\Leftrightarrow$). Let us now see how the CV value of compatibility is computed. For each step, we provide a running example.

Let S be a query which can be seen as a source component whose SD is composed of U weighted features ($S1(u)$, $S2(u)$), $u = 1, 2, \ldots U$, where $S1(u)$ and $S2(u)$ are the first and second term of the Uth feature, respectively. Let T be a target component described by V features ($T1(v)$, $T2(v)$, $v = 1, 2, \ldots, V$.

A weight is associated to T depending on the contexts specified in the query. Compatibility between S and T is computed by comparing feature terms, accessing the thesaurus to check synonymy values between them.

For the sake of brevity, our example considers the following query S

manage-graphics:	0.7
edit-text:	0.8

and a target component T

open-document:	0.2
insert-text:	0.4
draw-picture:	0.6

The comparison is carried out with respect to the position of terms in the feature, i.e., comparing the first and second terms among themselves only. Next, we compute two fuzzy relations represented by matrices called respectively *Equivalence* (EQ) and *Implication* (IMP). The dimensions of these matrices depend on the source SD S. These relations are fuzzy sets of features pairs, the first one taken from S and the second one taken from T. In the case of our example, IMP is the null 2×3 matrix.

As far as EQ is concerned, its (fuzzy) characteristic function is computed as follows:

$$f_{\mathrm{EQ}}(S(u), T(v)) = \min(\mathrm{SYNON}(S1(u), T1(v)), \mathrm{SYNON}(S2(u), T2(v)))$$

where SYNON is the matrix storing synonymy values. As the reader may notice, this relation operates conservatively, querying the thesaurus for synonymy values between features with pairwise-corresponding terms and taking the smallest return value. In other words, EQ is a fuzzy version of the classical crisp identity relation; if two terms are equal, it returns 1; if they are different (i.e., not interchangeable) it returns 0. Otherwise, it returns the minimum value read in the thesaurus. In our example, suppose the following synonym values hold (synonym terms are separated by commas):

manage,	open:	0.5
manage,	insert:	0.5
manage,	draw:	0.2
manage,	open:	0.3
manage,	insert:	0.3
manage,	draw:	0.6

Table I.

EQ	open-document	insert-text	draw-picture
manage-graphic	0.2	0.3	0.2
edit-text	0.3	0.3	0.3

graphics,	document:	0.2
graphics,	text:	0.3
graphics,	picture:	0.8
text,	document:	0.8
text,	text:	1
text,	picture:	0.3

then we obtain the EQ shown in Table I.

IMP, i.e., the fuzzy implication, can be computed as follows:

$$f_{\mathrm{IMP}}(S(u), T(v)) = \min(1, \max\{w(T(v)), w(S(u))\})[\mathrm{EQ}(u, v)]$$

where each w is the feature weight. According to IMP relation, interchangeability with respect to a feature is perfect if the required importance value for that feature is less or equal to the value found in the descriptor base. In our example, we obtain the relations shown in Table II.

Now, transposing IMP and right-multiplying it times EQ one obtains a matrix whose trace is called *satisfaction set*, or SAT. In the example, we obtain the following matrix:

0.1	0.114	0.1
0.114	**0.135**	0.114
0.1	0.114	**0.1**

SAT is as follows: {0.1,0.135,0.1}.

Being the result of the computation of a fuzzy implication, this set expresses the (fuzzy) interchangeability between S and T. SAT may itself be weighted by multiplying each element times the corresponding component of the normalized weights vector W', defined as follows:

$$W' = \frac{w(T(k))}{\sum_{i=1}^{V} w(T(i))}, k = 1, \ldots, V$$

The role of W' is to enhance the effect of matching of relevant (with high weight) features in the target component T during the computation of CV.

In our case, $W' = \{0.16,0.32,0.5\}$.

Thus we obtain a final set SIM which is again the result of the implication enhanced by a supplementary weighting. In the example, SIM $= \{0.016,0.043,0.05\}$.

Table II.

IMP	open-document	insert-text	draw-picture
manage-graphics	0.14	0.21	0.14
edit-text	0.24	0.24	0.24

By extracting a value from SIM by means of a *defuzzification function* D, one obtains the desired CV. In our model, defuzzification functions are attached to descriptors: each component is associated with its own function D. Our default choice for D is the triangular function, which is initially associated to all components. Its behavior is shown in Figure 9.

The values of SIM characteristic function, denoted by CV_1, CV_2, . . . , CV_K are arranged in increasing order and positioned at points $t = 1/K + 1, 2/K + 1, \ldots, K - 1/K + 1, K/K + 1$.

Now, CV is obtained as:

$$\mathrm{CV} = 10 \sum_{i=1}^{k} D_i CV_i$$

where 10 is a rescaling factor avoiding managing of small decimal values in the system. Obviously, being a uniform rescaling factor, it has no effect on the system operation. In our example, $K = 3$, and hence we obtain CV = 0.76.

A basic property that D should possess is to ensure *weak monotonicity* of CV with respect to the same query. This property can be stated as follows: given two components having the same description $SD_1 = SD_2$, with $w_{1k} \leq w_{2k}$ for $k = 1,2, \ldots, K$, where K is the number of features in these SDs, we must have:

$$\mathrm{CV}(SD_1) \leq \mathrm{CV}(SD_2)$$

in response to the same query.

This technique of defuzzification ensures weak monotonicity of CV. As an example, consider the values of D_i that we have used, i.e., 0.5, 1 and 0.5 obtained for $K = 3$. Recalling, that, as a result of the query we have the following values of SIM, $sim_{1,1} = 0.16$, $sim_{1,2} = 0.43$, $sim_{1,3} = 0.5$, now we consider a second component T', with the following SIM values: $sim'_{2,1} = 0.1$, $sim'_{2,2} = 0.3$, and $sim'_{2,3} = 0.4$. Computing the CV for both components T and T', we obtain $\mathrm{CV}(T') = 0.55 < \mathrm{CV}(T) = 0.76$.

The illustrated technique is suitable for code reuse, code understanding for restructuring or reengineering, or for search of services over a network of distributed providers. The basis for these applications is component retrieval, which allows the developers to analyze code bases in an effective and user friendly way. The information flow in a *retrieval system* is depicted in Figure 10.

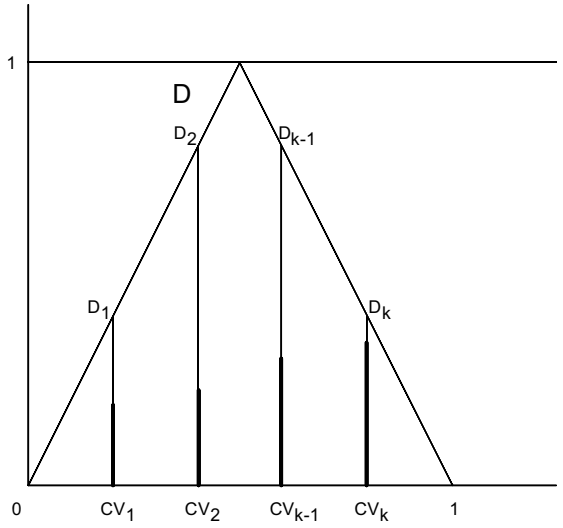


Fig. 9. The triangular defuzzification function D.

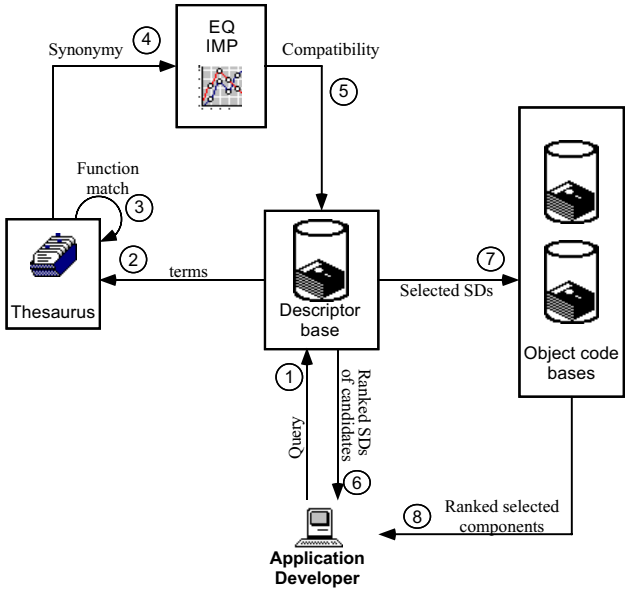


Fig. 10. Retrieval mechanism.

Queries submitted to the descriptor base are forwarded to the thesaurus where the SYNON matrix is consulted for synonymy values used to expand the user query with synonyms. Then, the query is expanded to include the new terms and is further processed, and the compatibility is computed. Selected SDs, representing (partially) matching components, are presented to the application developer and eventually used as input to the object code bases, for extraction of the suitable components.

4. TERM PREPROCESSING AND THESAURUS INITIALIZATION

Sometimes terms used in the code (such as variable and method names) are semantically poor and seem useless when included in a descriptor that should instead enhance code comprehension.

Therefore, a thesaurus is necessary, allowing terms to be made uniform through a naming discipline. Such a tool should support relationships among terms, such as *synonymy*, *broader term* (BT), *narrower term* (NT), *related term* (RT), etc., as defined in classical text retrieval thesauri [Salton 1971] in order to provide flexibility to the retrieval system.

In our opinion it is advisable to integrate an *automatic initialization phase* and a *manual filtering phase*.

Automatic support, based on a purely syntactic approach, has the advantage of considering the whole available vocabulary. However, obviously, it introduces a lot of *noise* (falsely related terms). Moreover, automatically adding semantics to terms seems a very difficult task even if more information about the code were available than it is usually provided.

On the other hand, building a thesaurus that relies only on human inspection of code results in an error-prone task that is very unlikely to be adopted in practice. The proposal described in this article can be outlined as follows: first, an automatic support for thesaurus construction extracts terms from the software descriptors, and preprocesses them in order to obtain a *controlled vocabulary*. Then, by exploiting contexts, a term frequency analysis is performed that allows us to compute a tentative synonymy matrix. Such a matrix is used to *initialize* the thesaurus and is the input for manual filtering. The purpose of the synonymy matrix is to maximize thesaurus recall (as will be defined in the Appendix).

The application engineer has the following tasks:

—filter out the false synonyms;

—classify the obtained synonyms as BT, NT, and RT.

Although long and tedious, these tasks are surely more feasible than building the whole thesaurus from scratch. Moreover, the proposed technique prevents forgetting terms or relationships. The thesaurus presented here is seen as a tool for *query expansion*, in order to avoid system silence. Although query expansion can increase noise, we believe noise here to be preferable to silence, since it can be then filtered using methods such as limiting the number of returned candidates. In this section, we show how the thesaurus is initialized, taking contexts into account (Section 4). The results of experimental validation of the thesaurus are reported in the Appendix.

4.1 Thesaurus Construction

Automatic thesaurus construction in a text retrieval system is traditionally performed by statistical analysis of term distribution, exploiting *filtering* and *clustering* techniques [Salton and Buckley 1988]. In software analysis, no true corpus of documents describing individual software components can

be assumed; moreover, accompanying documentation may be unfit for statistical analysis on term distribution. Hence we cannot rely on the statistical approach alone for thesaurus construction. Instead, we will exploit contexts, seen as facets of the classification of individual components. As discussed before, the SD of each component is partitioned in contexts representing the characterization in a specific application domain. Our basic assumption is that the cardinality of the jth partition in SD_i, i.e., the number of features belonging to the jth context in that SD_i hints to the relevance of the component to that particular context. Following a method outlined in Yu [1975], we construct a *term-context matrix*. The following *context relevance function* (CRF) initializes the matrix:

$$CRF_{l,j} = p_j(l)\frac{term_{l,j}}{term_l}$$

The relevance of the lth term in the jth context depends on the percentage of terms (*percentile* p_j) in the jth context having occurrences less or equal to those of the lth term in j, multiplied by the ratio between the occurrences of the lth term in the jth context and the total occurrences of the lth term over all contexts. This definition ensures that values of $CRF_{l,j}$ are not dependent on the relative context sizes. In fact, values depend on a percentile, i.e., a percentage of terms in the contexts, that can be the same even in contexts having a different number of terms. Moreover, $CRF_{l,j} \leq 1$ for every l, j. If a term never appears in a context, $CRF_{l,j}$ simply evaluates to zero.

The initialization procedure goes as follows:

—First a term-context matrix is built, having a row for each term and a column for each context. It has dimension $N_{term} \times C$. Our CRF will classify as relevant the terms corresponding to peaks in a profile, while compensating with respect to the relative context dimension. No provision is made at this level for the elimination of stop-words. In our opinion, a preliminary filter for stop-words is simpler and more effective than introducing "across contexts" dependency in CRF computation.

—Then, for each given value of $l(j)$ we interpret $CRF_{l,j}$, $1 \leq l \leq N_{term}$, and $1 \leq j \leq C$ as the characteristic function of a fuzzy set, obtaining two families of (possibly not disjoint) fuzzy sets. So we have *term sets*, corresponding to rows, and *context sets*, corresponding to columns. Here it is important to notice that there is no need for application engineers to deal explicitly with building this matrix; when adding a descriptor to the descriptor base, an SD editor may well update the vocabulary on the basis of the SD of the new component.

—Given two row vectors $(a_1, a_2, \ldots, a_j, \ldots, a_C)$ and $(b_1, b_2, \ldots, b_j, \ldots, b_C)$, representing terms A, B we define the rows representing $A \cup B$ and $A \cap B$ by means of the usual fuzzy sets operators.

—We are now ready to define m_j (*weight of a match*) as follows:

$$m_j = \frac{1}{1 + \Delta_j}$$

when both a_j and b_j are positive for $j = 1, 2. . . , C$ and zero otherwise. Δ_j is the difference $|a_j - b_j|$. Moreover, we define m_j^* (*weight of a mismatch*) as

$$m_j^* = \frac{\Delta_j}{1 + \Delta_j}$$

when either a_j or b_j are zero for $j = 1, 2, \ldots, C$. The mismatch weight is zero otherwise. When $a_j = b_j = 0$, neither a match nor a mismatch results, and both m_j and m_j^* are zero. So we have

$$M = \sum_{j=1}^{C} m_j$$

and

$$M^* = \sum_{j=1}^{C} m_j^*.$$

—By computing the fuzzy union of all contexts where each of our two terms A and B appear, and then computing matches and mismatches between resulting sets, we also obtain M' (total weight of context matches) and $M^{*'}$ (total weight of context mismatches).

—Next, we define the synonymy relation. A desirable property for this relation is to be *nonincreasing* in the number of context and terms mismatches and *nondecreasing* in the number of context and terms matches [Yu 1975]. Our fuzzy synonymy is defined as follows:

$$f = \min\left(1, \frac{1}{2}\left|\left(\frac{M - M^*}{M + M^*} + \frac{M' - M^{*'}}{M' + M^{*'}}\right)\right|\right)$$

for M^* or $M^{*'} \neq 0$. If $M^* = M^{*'} = 0$ and $A \neq B$, in order to avoid false synonyms, we set

$$f = \left|\frac{\frac{M}{n_{term}} - \frac{M'}{n_{ctx}}}{\frac{M}{n_{term}} + \frac{M'}{n_{ctx}}}\right|.$$

Although our thesaurus initialization technique exploits contexts, the resulting SYNON matrix is bidimensional, i.e., it is not itself faceted. Devising an alternative method in order to obtain a tridimensional matrix, thus adding a context dimension to the thesaurus, is theoretically possible, but in our opinion the additional computational burden makes the approach unpractical. We propose instead to take contexts into account while *using* the thesaurus, allowing a nonzero synonymy value to be returned only if the two terms appear in the same context for at least one of the contexts. Practical deployment of the above-described techniques suggested that synonymy values are not dynamically computed at query-processing time, but rather are saved in a long-term storage structure, called the SYNON matrix. SYNON has a slow-variating life cycle. Its update is done upon arrival of groups of terms, typically upon inclusion of a new library. Updates constitute a bordering of the original matrix. The comparison with the already present values is done under a *mod-min* interpretation (i.e., as the fuzzy intersection).

5. USER FEEDBACK

We intend to make our classification and retrieval system sensitive to user feedback. In our approach, user feedback is a long term learning process leading to permanent modification to the system in order to adapt it to the needs of the user community. In fact, our method is aimed at improving the overall performance of the classification system rather than optimizing a single query. Users are considered as *experts* whose opinions are polled simply by registering their choices. The superposition of expert opinions is a moot point in fuzzy weighting research [Bardossy et al. 1993]. In our model, *user feedback does not change weights*, but the *shape of the defuzzification function D* is modified as a consequence of users choices. User opinions polling is obtained as follows. Suppose that the ranked list of components has been presented to the user, and that the user does not select the first component in the list, but rather the component presented in the kth position. Then, all defuzzification functions $D(t)$ corresponding to components occupying positions from the first to the $(k - 1)$th will be affected by the following reshaping function, called *quality function* (QF):

$$D_{new}(t) = (1 - \beta)D_{old}(t) + \beta D_{corr}(t)$$

where $D_{corr}(t) = (1 + \gamma)t - \gamma$ for $t \leq 0.5$ and $D_{corr}(t) = 2(1 + \gamma)(1 - t) - \gamma$ for $t > 0.5$. In turn, the chosen component will be affected by the opposite reshaping, namely $D_{corr}(t) = 2(1 - \gamma)t + \gamma$ for $t \leq 0.5$ and $D_{corr}(t) = (1 - \gamma)(1 - t) + \gamma$ for $t > 0.5$.

Figure 11 shows depreciation and appraisal modification of the defuzzification function as a result of our user feedback mechanism. Parameter γ, which can be tuned by the application engineer, expresses system reactivity as well as the total variation scope. In fact, the higher is γ, the faster the system can modify the shape of the defuzzification function and the

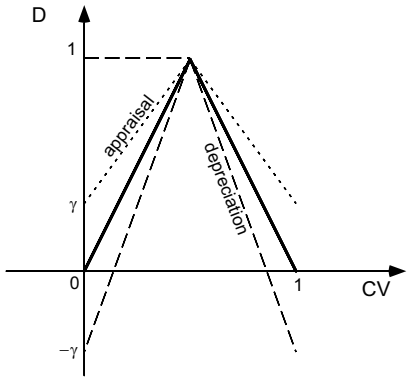


Fig. 11. The possible depreciation and appraisal of the defuzzification function D.

broader the total possible variations that can occur as a consequence of user feedback [Damiani and Fugini 1995]. The confidence granted to user opinion is expressed by parameter β, since variations of the function shape must only occur as a cumulative effect, normally $\beta \ll 1$. Thus, user feedback will cause a slow modification of the shape of the defuzzification function D, biasing it to reflect, in the fullness of time, the views of the user community about the reuse potential of the component it is associated to.

In order to set the value of β appropriately, the notion of *user* must be specified in more detail. First of all, even without taking into account the special role of the application engineer, not all application developers in the user community are at the same level of expertise; moreover their functions and responsibilities are not fixed but depend on the specific software process adopted at their site. The software engineering process is indeed a complex one, especially if a reuse policy is in force [Bellinzona et al. 1995] and can be organization, process or even product specific. The notion of user, seen as a part of the software development process, is a structured one. It depends on the process model adopted and may evolve in time following model evolution or change. Hence, we introduce the notion of a *user profile* as an indicator of a user's skills. This profile is a vector $\{\alpha_1, \alpha_2, \alpha_3\}$ of (possibly time-dependent) numerical values comprised between 0 and 1. These values express a user's *domain*, *task*, and *strategy* skills [Mi and Scacchi 1990], and depends on the development environment in terms of the currently adopted software process model, i.e., on the software process model currently in use. Task skill, for instance, can be measured by the number of development groups in which a user is involved; domain skill can be computed from the number of projects he is involved in the specific application domain, while strategy skill is related to his responsibility role in the team. Each time a user logs on to our system, skills must be aggregated in order to compute the single parameter β used in our feedback mechanism. The most general way of doing so is by using a generic *weighted average* (WA) operator, as follows:

$$\beta = H \sum_{i=1}^{3} w_i \alpha_i$$

with the constraint $\Sigma_{i=1}^{3} w_i = 1$. H is a bounding factor whose role is to keep $\beta \ll 1$. It must be set to the maximum value allowed for β; in our experimentation, we set $H = 0.1$. The weights w_i allow the application engineer to tune the feedback mechanism with respect to the composition of the development team which uses the system.

Suppose for instance the development team of a software house to be composed by two project managers leading a group of six senior and 12 junior programmers. The project managers' profiles show high strategy and domain but limited task skills ($\alpha_1 = 0.8$, $\alpha_2 = 0.1$, $\alpha_3 = 0.8$); senior programmers have moderate strategy skills, but high domain and task skills ($\alpha_1 = 0.8$, $\alpha_2 = 0.6$, $\alpha_3 = 0.2$) while juniors have adequate task skills but lack significant knowledge of domain and strategy ($\alpha_1 = 0.1$, $\alpha_2 = 0.8$, $\alpha_3 = 0.1$). It is reasonable to assume that project managers (and, up to a certain extent, senior programmers) can estimate the opportunity of reusing a component on the basis of their full awareness of the company's strategic reuse policy, which depends on several (external) factors, such as market and organizational issues. On the other hand, the number of junior programmers in the team could make their views prevail, potentially endangering the reuse policy. In order to compensate for this unwanted effect, the application engineer could privilege strategy and, to a lesser extent, domain skills by setting $w_1 = 0.3$, $w_2 = 0.2$, $w_3 = 0.4$. This would make $\beta = 0.058$ for project managers, $\beta = 0.044$ for senior programmers, and $\beta = 0.023$ for juniors. The same line of reasoning supports our *default setting* for weights w_i, which suggests to the application engineer weight values that outline skills which are less frequent in the development team. Namely, we set

$$w_i = \frac{\sum_{j \neq i} \sum_{k=1}^{n} \alpha_{j,k}}{2 \sum_{k=1}^{n} \alpha_{j,k}}$$

where n is the total number of users, and $\alpha_{j,k}$ is the jth skill value of the kth user. Reconsidering the above example, we see that this formula gives

$$w_1 = \frac{\sum_{k=1}^{20} \alpha_{2,k} + \sum_{k=1}^{20} \alpha_{3,k}}{2\left(\sum_{k=1}^{20} \alpha_{1,k} + \sum_{k=1}^{20} \alpha_{2,k} + \sum_{k=1}^{20} \alpha_{3,k}\right)}$$

$$= \frac{2(.1) + 6(.6) + 12(.8) + 2(.8) + 6(.2) + 12(.1)}{2(2(.8) + 6(.8) + 12(.1) + 2(.1) + 6(.6) + 12(.8) + 2(.8) + 6(.2) + 12(.1))}$$

$$= 0.34.$$

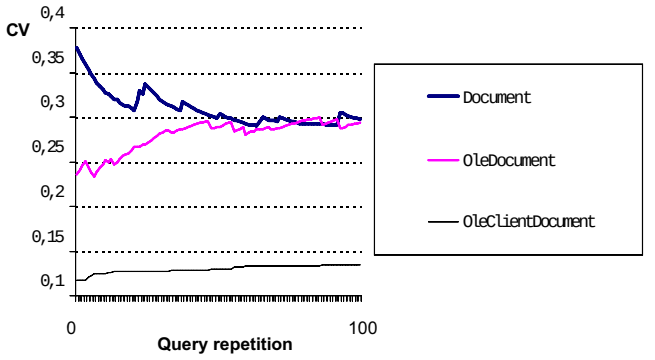


Fig. 12. Compatibility value (CV) changes as a consequence of user feedback (100 repetitions of the query).

The same computation for the other weights gives $w_2 = 0.23$ and $w_3 = 0.42$, so that in this case default values almost coincide with those estimated previously by the application engineer.

An idea of the cumulative effect of our user feedback mechanism is given by Figure 12, depicting the drift of the compatibility values of three components, `Document`, `OleDocument`, and `OleClientDocument` after 100 repetitions of query Q4 shown in Section 6. In our experimentation, 38 of the 100 queries were posed by project managers, 30 by senior, and 32 by junior programmers; all three categories of users showed a marked preference for `OleDocument` (chosen about 80% of the time), followed by `Document` (10%) and `OleClientDocument` (10%). As an effect of the users' attitude, the compatibility values of `Document` and `OleDocument` show respectively a sharp decrease (increase) and remain very close until an overtaking occurs. By biasing the defuzzification function D, we obtain the same effect on compatibility as the modification of features weights in the descriptors, but attain a flexibility with respect to the software process model.

5.1 User-Defined Query Semantics and Query Language

In our retrieval system query formulation requires the user to set up a simple list of the features the desired component should possess, together with their weights. Although this mechanism is very simple, this way of stating queries allows for a wide variety of query semantics. In our system, weights represent the *relative importance* of the listed properties. However weights could also be interpreted as *fulfillment degrees* of performance or other nonfunctional properties that must be guaranteed by the retrieved component. We explored the possibility of varying the query semantics according to the user needs in Bosc et al. [1998]. At the expense of simplicity, the retrieval environment can be endowed with a structured query language allowing the user to express the query semantics explicitly. This can be useful in some industrial applications and has been experi-

Table III. Characteristics of the Testbed Descriptor Base

	All	Generic	Graphic	Services	Interactive	Interface	EDT
Number of SDs	95	3	37	43	13	32	19
Minimum Length of SDs	1	5	1	1	1	1	1
Maximum Length of SDs	112	14	76	43	43	32	21
Average Length of SDs	15.8	9	9.78	7.95	15.76	10.9	11.36
Number of Features	1501	27	362	342	205	349	216
(Distinct)	771						
Number of Terms	464	20	235	223	162	216	70

mented for example in the Prosa system where a fuzzy component query language has been built on top of a relational database system holding SDs [Fusaschi and Montini 1997].

Weights are chosen in a query by the user according to the relative importance semantics. For example, in an initial formulation of a query, high weights are assigned to highlight the most relevant features, and lower weights can be used to keep the search space limited while including borderline, potentially useful components. Practical experience (see Damiani et al. [1997]) leads us to observe, that, users often formulate queries in the framework of query sessions, rather than posing standalone, occasional queries because the user has an explorative attitude and needs to evaluate several alternative reuse candidates. During query sessions, the user has two ways to modify the queries during the search: by varying the features and by varying the weights. By imagining these two variations as two axes, usually the developer will follow a path in the plane. The idea is to focus the search by adding or deleting features, and then to fine tune the retrieve set by means of the weights.

In general, adding features to a query limits the search space, while adding features with low weights serves to moderate this limitation, keeping some borderline component in the candidate component set. This usage pattern by developers corresponds to the above-mentioned attitude of search, especially useful against large components bases where components offering side functionalities also can be sometimes useful. Experience with the retrieval system shows that this usage pattern can be straightforwardly and easily understood and learned by reasonably experienced developers.

6. EXPERIMENTAL RESULTS

In this section, we present some experimental results obtained from a *tool suite* employing a set of fast C routines. This tool suite is intended as a rapid prototype aimed at testing the approach and not as a complete classification and retrieval facility. It is not aimed at testing the system usability nor the user friendliness of the interface of a potential prototype based on theses techniques (as done in other systems, for example in Henninger [1997]).

The overall procedure of code analysis, descriptor querying, and navigation follows the cycles depicted in Figure 13.

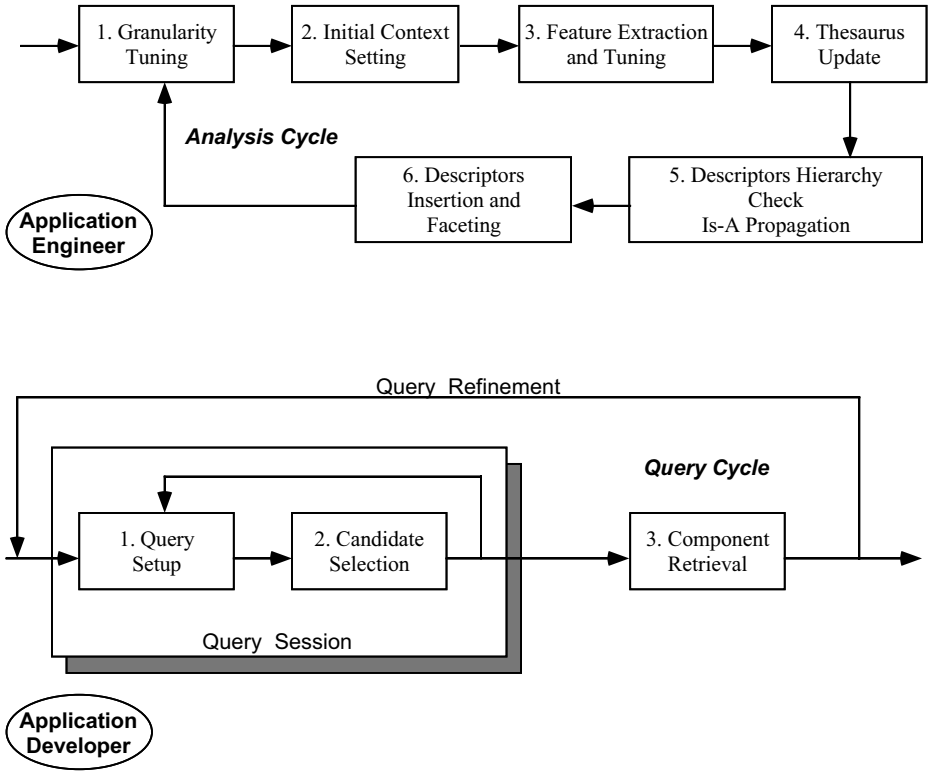


Fig. 13. Usage steps and user categories.

6.1 Code Base Construction

The base of code descriptors used by our tool suite consists of SDs obtained from the standard Visual C++ Foundation rel. 2.0 Class Library (version 3). This is a general-purpose library, not biased by a specific context but rather composed of heterogeneous classes providing a common ground for application development. Moreover, a well-known naming discipline exists in this library. The characteristics of the descriptor base are summarized in Table III (the "Services" context includes components for memory and file support, run-time support, and exception handling; the "EDT" context stands for "Extended Data Types").

In the experimental classification procedure the following choices were made:

(1) Each SD has the class name and includes features extracted from the class interface according to the method presented in the previous sections.

(2) The thesaurus is built from the whole base lexicon. Moreover, in the tool suite presented here synonymy values are computed only for verbs. This is due to the fact that, in our experience [Bellinzona et al. 1995; Fäustle et al. 1996], methods names tend to appear repeatedly in the

code base, while data structures are more local to classes and are hence less reused.

(3) In order to achieve a conservative result set, no manual filtering was performed on the obtained thesaurus.

Some results on the thesaurus intended to measure its quality and performance as a standalone tool were obtained on a larger code base and are reported in the Appendix.

6.2 Hierarchical Faceting

For the sample code base, C_{init} is set to "GRAPHIC" and "INTERFACE." Some classes in the hierarchy tree are classified also under other contexts; for some classes, features are assigned disjointly to different contexts by manual intervention.

The Is-a hierarchy of descriptors in the sample code base is depicted in Figure 14, where names of contexts are repeated in the descendants when the descendants exhibit some features of their own in that context. Referring to the SDs shown in the figure, for example, `CommandTarget` includes `Graphic` because it has its own features (e.g., "`start,cursor`") belonging to that contexts.

Inherited contexts are not shown: for instance, `Window Application` inherits the `Graphic` context which is not shown, while its own features are classified under the `Services` and `Interactive` contexts.

6.3 Query Formulation Protocol

In order to test our approach thoroughly, we have included in our suite a retrieval tool relying on the algorithms described in Section 3. Preliminary experiences with a previous version of the tool are reported in Fäustle et al. [1996]. Here we undertook an empirical evaluation aimed at enlarging and confirming those experiments and to test the effectiveness of the thesaurus.

Let us now describe how we performed the evaluation of the retrieval mechanism. We shall compare our results to those obtained from a baseline system based on the Unix `grep` family of commands (see Maarek et al. [1991]). As outlined in Section 5, the developer begins a query session by formulating a query composed of few obviously relevant features, and assigning to them weights based on relative importance semantics [Damiani and Fugini 1997]. The user can proceed in the search session by varying the features and/or weights.

6.3.1 *Query Setup*. This first set of tests does not include the thesaurus, in order to test the effectiveness of the retrieval mechanism alone and to obtain numerical values for recall and precision.

The experimental set of queries has been defined using the same experimental protocol used in Fäustle et al. [1996], which was now executed against the new component base. We have set up three blocks of five

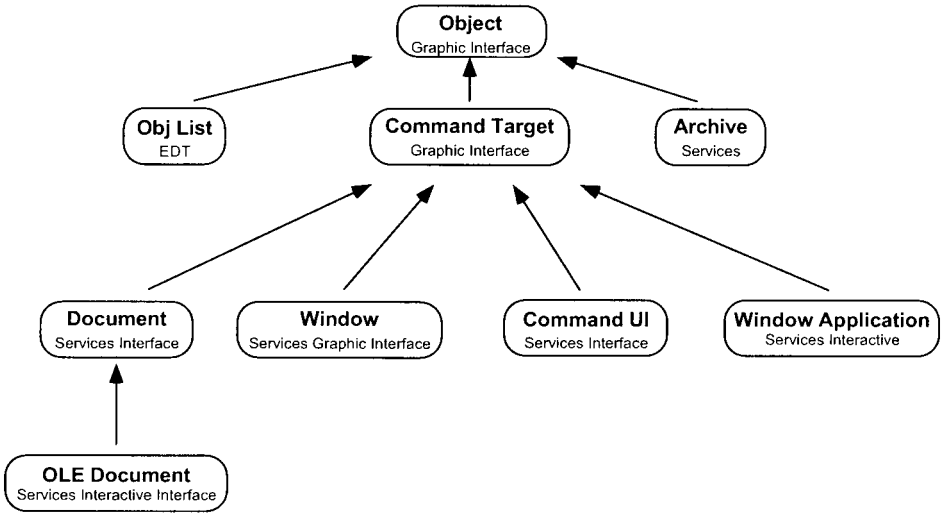


Fig. 14. Partial Is-a hierarchy of the code base.

queries each and executed them against the component base. The total number of queries is comparable to the number used in Maarek et al. [1991], given the size of our repository. In order to obtain a meaningful number of runs, the queries have been executed each three times, at different threshold values for the ranked returned list. Our test queries were prepared off-line and then submitted to the system in succession. No substitution or reformulation of queries on the basis of their results is allowed. A different approach was undertaken in the evaluation of the compared baseline system, where query feedback is allowed on query formulation.

Within each block, the queries are related in order to reproduce the human behavior outlined in Section 5. Each block is a retrieval session where the user progressively refines an initial query to search for a certain kind of component.

Each query was submitted to the judgment of a human expert acquainted with the component base who marked the components which should have been retrieved. These components constitute the query control set. The same query was then submitted to the system.

As in Fäustle et al. [1996] and Maarek et al. [1991], recall and precision are defined as follows:

$$recall_T = \frac{\#\ of\ retrieved\ components\ belonging\ to\ the\ control\ set}{cardinality\ of\ the\ control\ set}$$

$$precision_T = \frac{\#\ of\ retrieved\ components\ belonging\ to\ the\ control\ set}{\#\ of\ retrieved\ components}$$

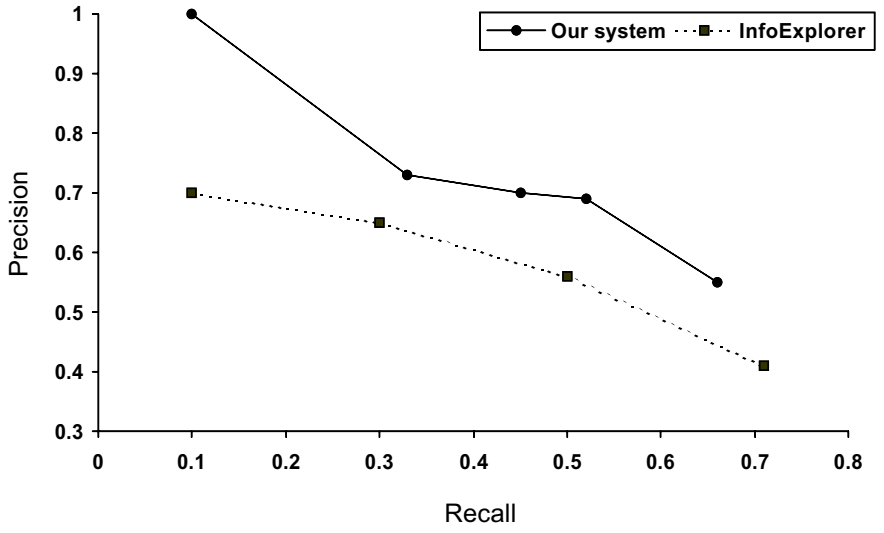


Fig. 15. Precision and recall of the test set at 0.5, compared with precision and recall of a baseline system.

6.3.2 *Query Execution*. For each query block B_i, $i = 1, \ldots, 3$ we executed the queries $Q_{i1}, \ldots, Q_{i3}$ and computed the recall values ($r_{i1}, \ldots, r_{i3}$) and the precision values ($p_{i1}, \ldots, p_{i3}$). Each block has been executed three times, with threshold values 0.1, 0.4, and 0.5 respectively for the ranked list. A sample query block is reported together with its precision and recall values in the Appendix.

Then, we ordered the recall-precision pairs for increasing values of recall, taking the median of precisions for the same recall value. This is repeated for the three threshold values. Across query blocks, we averaged the precision values for each obtained recall value. This procedure is analogous to the procedure outlined in Maarek [1991].

The results of precision and recall for the 0.5 threshold are plotted in Figure 15. Also, in the figure the results of the baseline, grep-based system Infoexplorer [IBM 1990] are shown as a basis for comparison only, since the results of that system have been obtained by selecting the best results out of a much wider set of queries.

We observe that the system shows a good behavior at the lower/upper limits of the considered range of recall values, while keeping satisfactory precision values in the central portion of the range.

6.4 Usage of the Thesaurus

Since the purpose here is to test the impact of the synonymy relationship, we use a set of test based on single-query probes. Query sessions with the thesaurus are conducted as follows. Queries are formulated by an application developer, aware of the application requirements, of the available

contexts and of the system lexicon, but with only the knowledge of the list of available component names. The developer is also helped by the verb-noun semantics, in that when he or she enters a verb the corresponding list of names which are actually coupled to it in the descriptors are proposed. On the other hand, the developer is unaware of synonymy values.

Five basic queries are formulated. Each is repeated on all the possible combinations of the contexts chosen by the user plus the default execution over the entire descriptor base. The total queries are 28, each submitted twice, respectively without and with thesaurus support. Hence, the total number of queries, given the size of our code base, is comparable with the size of test suites used for the validation of other systems. It is important to underline, that, in order to obtain a conservative evaluation, the system is tested shortly after completion of the classification procedure. So, recall and precision are measured without the support of the adaptive system.

The queries are the following:

(Q1) This query is aimed at retrieving an object of type Window to edit text documents. The query is

enable-window:	0.8
edit-text:	0.8
select-text:	0.5
find-text:	0.7

submitted under the `Interface`, `Services`, and `Interactive` contexts. Q1 is therefore executed 8 times without the thesaurus and 8 times with the thesaurus enabled. The control set of Q1 is {Window, View, ScrollView, EditView, EditWindow, FormView, FrameView}.

(Q2) This query is aimed at retrieving an exception handler. The query is

implement-exception:	0.8
cause-exception:	0.7
process-exception:	0.8

submitted under the `Services` context. Q2 is therefore executed 2 times without the thesaurus and 2 times with the thesaurus enabled. The control set of Q2 is {MemoryException, Exception, FileException, ArchiveException, UsrException, OleException, NotSupportedException, ResourceException, CommandTarget}.

(Q3) This query is aimed at retrieving a ToolBar object that can be decorated by user-defined bitmaps. The query is

set-button:	0.8
create-bitmap:	0.5
handle-bitmap:	0.5
draw-icon:	0.4

submitted under the `Interface`, `Interactive`, and `Graphic` contexts. Q3 is therefore executed 8 times without the thesaurus and 8 times with the thesaurus enabled. The control set of Q3 is {ToolBar, Menu, FrameView}.

(Q4) This query is aimed at retrieving a document manager object. The query is

open-document:	0.7
save-document:	0.7
modify-document:	0.8
add-item:	0.5

submitted under the `Interface`, `Interactive`, and `Services` contexts. Q4 is therefore executed 8 times without the thesaurus and 8 times with the thesaurus enabled. The control set of Q4 is {Document, DocTemplate, OleDocument, OleServerDoc, OleClientDoc}.

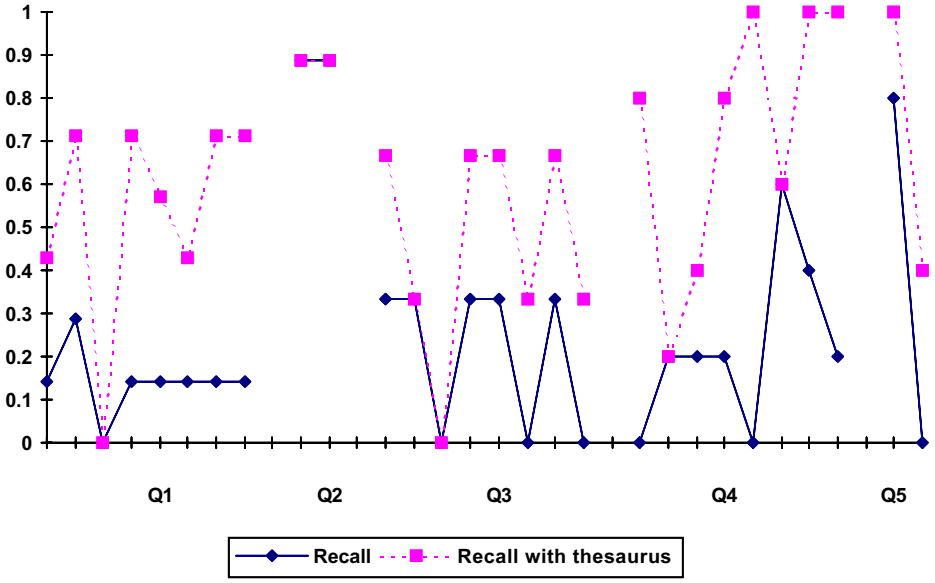
(Q5) This query is aimed at retrieving a resizable array class. The query is

access-index:	0.8
get-bound:	0.3
get-size:	0.3
set-size:	0.3

submitted under the `EDT` context. Q5 is therefore executed 2 times without the thesaurus and 2 times with the thesaurus enabled. The control set of Q5 is {Byte Array, Double Word Array, Object Array, Ptr Array, String Array}.

6.5 Results

The results of query execution are presented in Figure 16, where the values of recall and precision are reported for the five queries. The above results clearly show that thesaurus enabling causes a notable increase in recall, at least doubling it but in other cases increasing it as much as four times. As one would expect, this increase is usually obtained at the expense of precision (with the exception of Q5). This *noise* effect, however, is largely due to the fact that the thesaurus was used "as initialized," without any manual filtering. Such an increase is not always obtainable, as shown by the results of Q2, where the query formulation results are focused with respect to the code base, and hence the control set is retrieved at the first run. However, thesaurus enabling does not worsen the precision value that much. This suggests that the thesaurus usually enhances the retrieval effectiveness, and only leaves the values unaltered when the query is properly focused, e.g., because the queries are issued by users who are expert of the domain. Practical experience with the system shows that accurate thesaurus tuning is essential to limit noise.



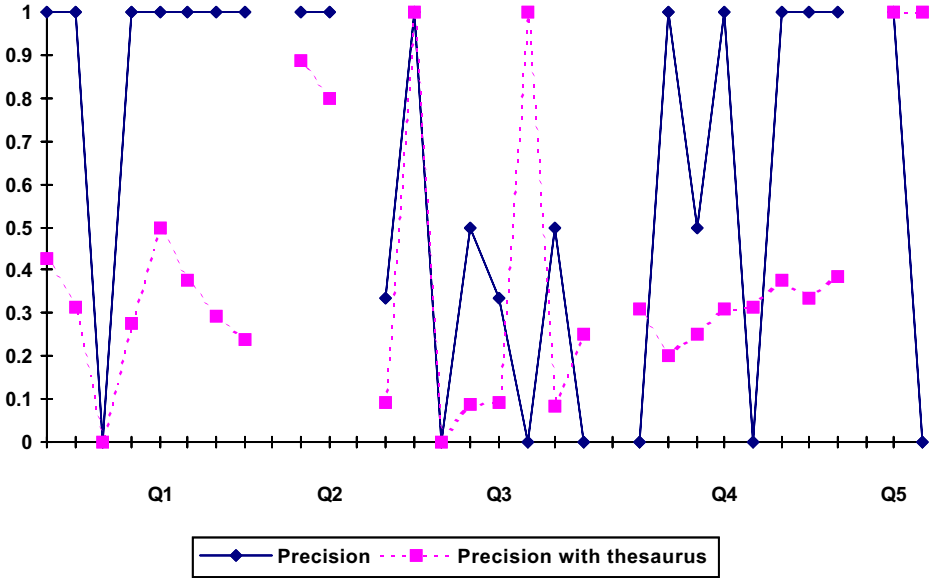


Fig. 16. Recall and precision values without and with the thesaurus.

7. RELATED WORK

The two related but distinct problems of component *classification* and *selection* of suitable components have been first dealt with in the domain of

repository-based *software reuse*, itself a much investigated subject [Banker et al. 1993]. Context-oriented libraries, such as mathematical libraries [Hopkins and Phillips 1988], were a first basic solution to the classification aspect. From early *flat* classification schemes, libraries later evolved to hierarchical faceted schemes [Prieto-Díaz 1991]. Object-orientation and the availability of class libraries brought further developments: the semantic of reusable objects is now commonly used in object-oriented libraries, such as in Eiffel [Meyer 1990] and Delphi [Cantù 1996]. Richer semantic models, and more structured organizations (e.g., including refinement levels and views), are proposed for example in AT&T's LaSSIE [Devanbu et al. 1991], in ConceptBase [Jarke 1993], and Lin [Batini et al. 1993], while software patterns are proposed as a standard representation of successful solutions to common software problems [Schmidt et al. 1996]. Reuse at the level of code specification is often based on the concept of component matching; a complete survey about *specification matching*, i.e., the process of determining if two software components are related, can be found in Moormann Zaremski and Wing [1997].

Organization of software in hypertext systems with the purpose of enhancing the visual interaction has been applied to a number of systems [LeVan 1996; Mi and Scacchi 1990; Jarke 1993]. The idea of storing (and/or using for classification) software artifacts, such as development documents, has also been proposed to increase reuse effectiveness [Batory and O'Malley 1992]. From the retrieval point of view, the components selection aspect of all those systems relied mainly on information retrieval and database-oriented techniques. Selection/retrieval methods can be classified as *formal methods*, *database-oriented* retrieval (Cactis of the Arcadia environment [Taylor et al. 1988], PCTE [Boudier et al. 1988], Vague [Motro 1988]), *knowledge-based* retrieval [Devanbu et al. 1991; Mi and Scacchi 1990; Ostertag et al. 1992], *hypertext navigation* (e.g., DIF and CHARLIE [Mi and Scacchi 1990; LeVan 1996]), and finally as *general-purpose approaches* combining some of the above methods. Other approaches try to compute *similarity coefficients* to better focus the search [Ostertag et al. 1992]; for example, an approach based on the analogy paradigm is presented in Maiden [1991]. The similarity between the concepts of two classes is often quantified in terms of *conceptual distance*. Such a metric is used to minimize the conceptual entropy of class hierarchies which undergo frequent changes; for example, in Dvorak [1994], the conceptual distance is a metric to measure the similarity among Smalltalk classes and to run a subclass restructuring algorithm. Another technique based on analogous reuse is presented in Spanoudakis and Constantopoulos [1994], with special attention on how to capture similarity between software artifacts organized in a conceptual repository. The signature-matching approach [Moormann Zaremski and Wing 1995] appears to be especially suited for code-level components, as it tries to exploit type information to compute a matching function between them. In Moormann Zaremski and Wing [1997], a complete survey of *specification matching* for software components is presented as the process of determining if two software components are

related, for a number of application development purposes, such as retrieval, reuse, substitution, and subtype identification. A list of tasks, besides retrieval, which can benefit from a reuse system is described in Sen [1997]. In several of the above approaches, the classification/retrieval problem is tackled in the domain of a repository, seen as a part of a development information system [Bellinzona et al. 1995].

Several contributions propose quick-and-simple keyword-based classification techniques, relying on term extraction from source code or artifacts. In Maarek et al. [1991], terms are extracted from software using an information retrieval approach applied to program comments, but the classification system is not hierarchy aware. In Paul and Prakash [1994], who tackle the issue of code search for maintenance, the approach is pushed further in order to attain a mechanism to understand what the code does. The approach employs a specification language, embedded in the SCRUPLE tool, to describe and to pose queries about a program's structure. Techniques for document analysis for automatic knowledge acquisition are also related work; for example, in Tang et al. [1994], a technique is proposed to extract design knowledge that can help in program understanding.

While our approach may be described under the above framework, we are even more interested in abstracting features and describing their relevance in order to express code behavior, thus enhancing code comprehension and improving domain modeling. Not all the above-described approaches appear to be well suited to this more general aim. Our protocol for extracting meaningful features with controlled granularity in order to describe object-oriented code, besides retrieving adaptable components, is intended to help the developer to understand the history of past developments. In both respects, fuzzy weights give flexibility to the system: code components can be easily described in terms of more or less relevant (*dominant*) characteristics, while our fuzzy calculus substitutes reasoning as used in Spanoudakis and Constantopoulos [1994].

It is interesting to remark that our features, extracted from source code, might well be complemented with others, describing different properties of software (such as its static structure as represented by a compiler, and/or its dynamic properties as seen by a test execution) during the classification and retrieval phases. For instance, the approach described in Podgurski and Pierce [1993] is based on the execution of reusable software; others capture the functionalities of components by using specifications during the retrieval process, thus addressing the aspect of precision in the retrieval. An example is the PARIS system [Katz et al. 1987], where specifications drive the retrieval. A more recent example is described in Mili et al. [1997], where refinement ordering between specifications is used to drive the selection of programming solution.

Another moot subject is how to deal with the system vocabulary. To this respect, we observe that a well-established technology already exists: thesauri have long since been used as a support for text retrieval, especially in large document bases. In fact, querying mechanisms based on string matching, however efficient, may prove completely unsatisfactory if

users do not know precisely which terms to use when formulating queries. Using the "wrong" (even if semantically equivalent) terms in queries may prevent the retrieval of documents the user would be interested in.

Such difficulties can lead users to rapidly loose confidence and, in the end, to abandon the system altogether. Conventional thesauri, i.e., collections of semantically equivalent terms, may relieve this problem but are expensive and slow to build and maintain by hand. So, many attempts were made, with mixed results, to automate their construction.

The basic assumption of computer-aided thesaurus construction for text retrieval systems is that differences between the statistical frequency distribution characteristics of terms in the overall document base and the frequency distribution of the same terms in a retrieval subset are, to a great extent, directly related to semantic factors [Yu 1975; Salton 1971; Salton and Buckley 1988].

First, the vocabulary of terms is filtered to retain only *relevant* ones and then equivalence classes with respect to a suitable relation between terms are computed.

Roughly, this relation (*cooccurrence*) is computed by weighting the number of times two relevant terms appear together in the document base against the number of times they appear by themselves.

Then, considering the Cartesian product of the filtered vocabulary of terms, a binary matrix is built, setting to 1 entries corresponding to couples of terms whose cooccurrence is above a threshold value, and setting to 0 entries corresponding to couples of terms whose cooccurrence is below the threshold.

Determining a way to compute this threshold other than by trial and error is still an unsolved problem in text retrieval research. Clustering techniques consider this matrix as the adjacency matrix of a graph whose nodes represent terms. Then, by computing all complete subgraphs (*cliques*) of this graph, the desired equivalence classes are obtained. Clusters partition the vocabulary in a (possibly not disjoint) family of crisp subsets. While surely unfit for constructing ready-to-use thesauri, this approach was used with not completely unsatisfactory results to build "quick and dirty" specialized thesauri to be later manually filtered by experts of the field [Salton 1971].

Considering software components as text documents, and refraining ourselves to thesaurus initialization, we believe that some version [Salton et al. 1994] of automatic construction techniques can be applied. We leave to manual filtering the task of improving the thesaurus quality at a satisfactory level; the filtering effort should however be considerably less than building a thesaurus from scratch. Anyway, text retrieval techniques need to be specialized in order to automatically initialize a thesaurus on the lexicon used in software components. In fact, statistical analysis of term distribution is not suitable for source code, since the copresence of terms (language terms, variable names, etc.) does not imply any semantic relation. However, some semantic relations may be spotted by considering

contexts and the relevance of terms in that context. This is particularly true for object-oriented code, where class names and methods usually have names semantically relevant for their description [Etzkorn and Davis 1997].

8. CONCLUDING REMARKS

In this article, we have described a suite of conceptual techniques for software classification and retrieval based on the use of descriptors stored in a structured repository. Descriptors are employed as compact analysis documents which can be handled more effectively than components themselves while retaining most of the information about components and their relationships in the application. The code classification model described in the article relies on a faceted, hierarchy-aware technique and on thesaurus support for the analysis of object-oriented code bases. Synonymy is used to further expand the model flexibility, in that the code base can be searched also for candidates that "resemble" the needed one because they are described with synonym terms. Experimental evidence is provided to validate the system and to show the results of thesaurus enabling. A mechanism of user feedback is outlined, aimed at tuning the system according to the views of the repository users.

Many of the ideas described in the article are suited for immediate implementation while others need to be adapted to specific application environments. Currently, the proposed suite of techniques is being used as the basis of some industrial repository-based environments. For example, in Fusaschi and Montini [1997] the whole basis of techniques has been integrated into an industrial large-scale development environment (more than two million lines of object-oriented code in the last three years). Namely, the repository has been implemented on a relational database endowed with a Web front-end. In that system, a component query language (CQL) has been introduced as a fuzzy extension to conventional SQL; linguistic variables are used instead of numerical weights, to facilitate user interaction. The user feedback part has been adapted to include a reward mechanism aimed at encouraging reuse through monetary compensation to developers.

The characteristic of providing conceptual toolkits to be later adapted and implemented depending on the needs of an organization is shared with many other research approaches. For example, a retrieval system named Selection Tool has been implemented during the EEC Ithaca project [Bellinzona et al. 1995] embedding the described classification and retrieval mechanism. The purpose of the prototype has been to perform a first evaluation of the approach using the Telos knowledge representation language as a host environment, and studying the interface of the mechanisms with the Ithaca tools environment. The Telos prototype, internally, converts the symbolic values of weights at the user interface—belonging to the set `VeryHigh`, `High`, `Medium`, `Low`, `VeryLow`—into numeric values, corresponding to fuzzy values. Similarly, the CodeFinder system presented

in Henninger [1997] is defined as a design-level prototype of a retrieval system explicitly meant to suggest useful solutions to software repository implementers and designers.

The definition of toolkits of techniques proves useful in a wide range of applications. The presented approach can be employed to organize software artifacts for code comprehension, reverse engineering, and domain modeling, analogously to the applications described in Sen [1997] and Moormann Zaremski and Wing [1997]. In fact, thanks to contexts and to the control of classification granularity, the descriptor base can be seen as a first basic representation of the application domain knowledge. Moreover it can support systematic exploration of large and distributed repositories of components in order to identify desired services on the basis of functional and nonfunctional information about them. Some of these aspects have been presented here; on other aspects of our research we are currently investigating, such as on run-time server identification over a network. Regarding this last issue, we have developed a prototype of a Trader for run-time selection of CORBA Servers [Bosc et al. 1998].

In general, we are aware that software artifacts classification has a cost in terms of time and resources. However, the presented techniques aim to minimize the effort required to set up and maintain the classification system because they rely on automatic tools whenever possible. Moreover, when properly coupled with a navigational environment, the proposed system can efficiently support both querying and browsing in order to get acquainted with the content and organization of code bases. Hence, the system can be smoothly integrated with a development environment.

APPENDIX

A. SYSTEM AND THESAURUS VALIDATION

We show a sample query block prepared and tested using the protocol described in Section 6.

In this retrieval session, the developer is looking for component(s) managing graphical windows (see Table IV).

In the graph of Figure 17, we show the results of the block of queries in terms of precision and of recall.

We are now ready to describe the thesaurus validation procedure.

In the experimental evaluation of our techniques for thesaurus initialization, we processed several libraries of object-oriented code, namely the NIH library [Gorlen et al. 1990] of extended data types, the Windows library of graphical objects of Borland C++ compiler rel. 4.0, and the Visual C++ rel. 2.0 class library. In order not to be biased toward any particular programming language we also examined some Smalltalk and Eiffel libraries. We obtained a total of about 150 SDs. This analysis led us to the following considerations:

—*Thesaurus terms:* The controlled vocabulary includes 90 terms, thus turning out to be small enough to be easily handled by the system and

Table IV.

Q1	
activate-window:	0.8
destroy-window:	0.8
create-window:	0.8
Q2	
activate-window:	0.8
destroy-window:	0.8
create-window:	0.8
build-multidoctemplate:	0.5
activate-mdichildwindow:	0.5
Q3	
activate-window:	0.8
destroy-window:	0.8
create-window:	0.8
build-multidoctemplate:	0.5
activate-mdichildwindow:	0.5
show-scrollbar:	0.8
set-background:	0.3
Q4	
activate-window:	0.8
destroy-window:	0.8
create-window:	0.8
build-multidoctemplate:	0.5
activate-mdichildwindow:	0.5
show-scrollbar:	0.8
set-background:	0.3
tile-mdiframewindow:	0.3
resize-parent:	0.5
Q5	
activate-window:	0.8
destroy-window:	0.8
create-window:	0.8
build-multidoctemplate:	0.5
activate-mdichildwindow:	0.5
show-scrollbar:	0.8
set-background:	0.3
tile-mdiframewindow:	0.3
resize-parent:	0.5
attach-window:	0.3
detach-window:	0.3

easy for the users to get acquainted with. This in spite of the relatively large number of classes which have been analyzed. Filtering was limited to the minimum and consisted in eliminating programming language keywords such as elementary type names. Moreover, we applied the verb-noun paradigm.

—*Influence of the number of samples over lexical quality:* We observed that an increase of the number of analyzed components leads to a better quality of the context relevance semantics. This appears to be true under the hypothesis that the lexicon of the system eventually reaches a stable size. In fact, in an operational setting of the descriptor base it is likely

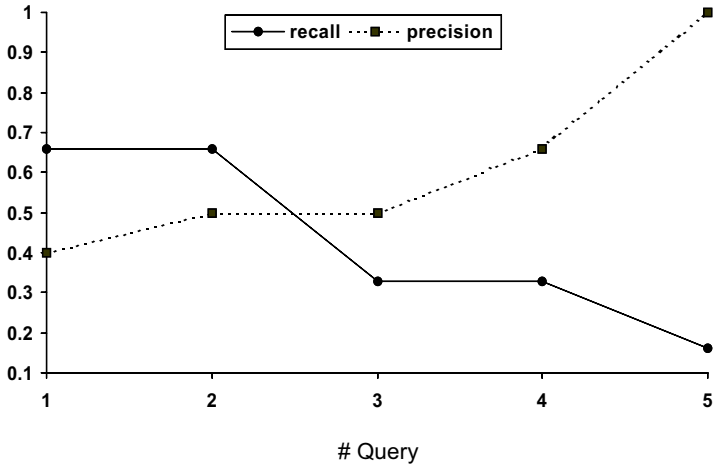


Fig. 17. Precision and recall of query block.

that components refer to a few, related contexts. Thus, upon arrival and assimilation of new libraries, the associated lexicon usually evolves toward a sufficiently rich and expressive domain vocabulary. So, updates made after a certain (domain-dependent) stability threshold of the lexicon has been reached can be considered as performed with constant lexicon.

—*Synonymy quality assessment:* We measured the quality of our thesaurus initialization employing the following procedure. We randomly chose 10 terms and had a human expert mark their synonyms blindly on the controlled vocabulary. These sets of synonyms are called the *control sets* associated to each term. Next, we considered the rows of the SYNON atrix corresponding to the terms and computed the following two parameters at two different threshold values T of synonymy:

$$recall_T = \frac{\#\ of\ retrieved\ synonyms\ belonging\ to\ the\ control\ set}{cardinality\ of\ the\ control\ set}$$

$$precision_T = \frac{\#\ of\ retrieved\ synonyms\ belonging\ to\ the\ control\ set}{\#\ of\ retrieved\ synonyms}$$

These parameters are tailored versions of the classical precision and recall measures used in information retrieval. Table V shows the precision and recall values for some terms randomly extracted from the thesaurus at different threshold values.

The average values of recall and precision at the 0.70 threshold are 0.75 and 0.20 respectively. We observe that the 0.70 value was chosen for the lower threshold because even lower values, while pushing the recall to 1,

Table V. Values of Precision and Recall for Randomly Chosen Terms of the Thesaurus

Term	Threshold	Recall	Precision
answer	0.80	0.3	0.13
	0.70	**0.81**	**0.16**
compare	0.80	0.4	0.15
	0.70	**0.4**	**0.15**
array	0.80	0.5	0.5
	0.70	**0.78**	**0.35**
edit	0.80	0.2	0.28
	0.70	**0.7**	**0.2**
arrange	0.80	0.28	0.15
	0.70	**0.71**	**0.18**
create	0.80	0.16	0.2
	0.70	**1**	**0.18**
remove	0.80	0.31	0.18
	0.70	**0.75**	**0.15**
size	0.80	0.4	0.3
	0.70	**0.8**	**0.25**
window	0.80	0.45	0.3
	0.70	**0.82**	**0.21**
write	0.80	0.28	0.15
	0.70	**0.7**	**0.2**

lead to virtually 0 precision. This would obviously be equivalent to select synonyms manually. On the other hand a high value of the threshold ($>$ 0.8), while possibly slightly increasing precision, excludes many true synonyms. This effect happens because terms used in exactly the same contexts are not necessarily the best synonyms even if they are considered semantically related by our CRF. Anyway, the 0.70 threshold filters out on average the 90% of the terms, allowing the application engineer during the subsequent tuning phase to select the real synonyms in a restricted set of candidates instead of browsing the whole vocabulary. Thus, this method proved to be an effective initialization technique.

Of course, the threshold tuning must be done for the lexicon pertaining to each context, and should be recomputed upon substantial system updating.

ACKNOWLEDGMENTS

We thank the partners in the project "Progetto Coordinato Ambienti di Supporto alla Progettazione di Sistemi Informativi" for useful discussions.

We also want to thank the anonymous referees of this article for their useful suggestions.

REFERENCES

BANKER, R. D., KAUFFMAN, R. J., AND ZVEIG, D. 1993. Repository evaluation of software reuse. *IEEE Trans. Softw. Eng. 19*, 4 (Apr.), 379–389.

BARDOSSY, A., DUCKSTEIN, L., AND BOGARDI, I. 1993. Combination of fuzzy numbers representing expert opinions. *Fuzzy Sets Syst. 57*, 2 (July), 173–181.

BASILI, V. R. AND ROMBACH, H. D. 1991. Support for comprehensive reuse. *Softw. Eng. J. 6*, 5 (Sept. 1991), 303–316.

BATINI, C., DI BATTISTA, G., AND SANTUCCI, G. 1993. Structuring primitives for a dictionary of entity relationship data schemas. *IEEE Trans. Softw. Eng. 19*, 4 (Apr.), 344–365.

BATORY, D. AND O'MALLEY, S. 1992. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol. 1*, 4 (Oct. 1992), 355–398.

BÄUMER, D., GRYCZAN, G., KNOLL, R., LINIENTHAL, C., RIEHLE, D., AND ZÜLLIGHOVEN, H. 1997. Framework development for large systems. *Commun. ACM 40*, 10 (Oct.), 53–59.

BÄUMER, D., KNOLL, R., GRYCZAN, G., AND ZÜLLIGHOVEN, H. 1996. Large scale object-oriented software-development in a banking environment: An experience report. In *ECOOP'96—Object-Oriented Programming, 10th European Conference* (Linz, Austria, July), P. Cointe, Ed. Springer-Verlag, New York.

BELLINZONA, R., FUGINI, M. G., AND PERNICI, B. 1995. Reusing specifications in OO applications. *IEEE Softw. 12*, 2 (Mar.), 65–75.

BIGGERSTAFF, T. J. AND PERLIS, A. J., Eds. 1989. *Software Reusability: Vol. 1, Concepts and Models*. ACM Press, New York, NY.

BIGGERSTAFF, T. J., MITBANDER, B. G., AND WEBSTER, D. E. 1994. Program understanding and the concept assignment problem. *Commun. ACM 37*, 5 (May 1994), 72–82.

BOOCH, G. 1994. *Object-Oriented Analysis and Design with Applications*. 2nd ed. Benjamin-Cummings Publ. Co., Inc., Redwood City, CA.

BOSC, P., DAMIANI, E., AND FUGINI, M. G. 1998. Dynamic service identification in a CORBA like environment. In *Proceedings of the 1st International Workshop on Innovative Internet Information Systems* (Pisa, Italy, June), D. M. Schwartz, Ed.

BOUCHON-MEUNIER, B., RIFQI, M., AND BOTHOREL, S. 1996. Towards general measures of comparison of objects. *Fuzzy Sets Syst. 84*.

BOUDIER, G., GALLO, F., MINOT, R., AND THOMAS, I. 1988. An overview of PCTE and PCTE. *SIGPLAN Not. 24*, 2 (Feb.), 248–257.

CANTÙ, M. 1996. *Delphi 2.0*. McGraw-Hill, Inc., New York, NY.

CASHIN, P. 1991. Bnr remains at the fore front of computing technology. *Telesis 92*.

CASTANO, S. AND DE ANTONELLIS, V. 1993. A constructive approach to reuse of conceptual components. In *Proceedings of 2nd ACM/IEEE International Workshop on Software Reusability* (Lucca, Italy, Mar.). ACM, New York, NY.

COX, B. J. 1986. *Object-Oriented Programming—An Evolutionary Approach*. 1st ed. Addison-Wesley, Reading, MA.

DAMIANI, E. AND FUGINI, M. G. 1995. Automatic thesaurus construction supporting fuzzy retrieval of reusable components. In *Proceedings of ACM SIG-APP Conference on Applied Computing* (SAC '95, Nashville, Feb.). ACM Press, New York, NY.

DAMIANI, E. AND FUGINI, M. G. 1997. Fuzzy identification of distributed components. In *Computational Intelligence—Theory and Applications*, B. Reusch, Ed. Springer Lecture Notes in Computer Science, vol. 1226. Springer-Verlag, New York, 550–552.

DAMIANI, E., FUGINI, M. G., AND FUSASCHI, E. 1997. A descriptor-based approach to OO code reuse. *IEEE Computer 30*, 10 (Oct.), 73–80.

DEVANBU, P., BRACHMAN, R., AND SELFRIDGE, P. G. 1991. LaSSIE—A knowledge-based software information system. *Commun. ACM 34*, 5 (May), 34–49.

D'SOUZA, D. 1996. Java: Design and modeling opportunities. *J. Obj. Orient. Program. 9*, 5 (Sept.).

D'SOUZA, D. F. AND WILLS, A. C. 1997. *Catalysis: Component and Framework-Based Development*. Addison-Wesley, Reading, MA.

DVORAK, J. 1994. Conceptual entropy and its effect on class hierarchies. *IEEE Comput. 27*, 6 (June 1994), 59–63.

ETZKORN, L. H. AND DAVIS, C. G. 1997. Automatically identifying reusable OO legacy code. *IEEE Computer 30*, 10 (Oct.), 66–71.

FÄUSTLE, S., FUGINI, M. G., AND DAMIANI, E. 1996. Retrieval of reusable components using functional similarity. *Softw. Pract. Exper. 26*, 5, 491–530.

FUSASCHI, E. AND MONTINI, A. 1997. The ESSI/PROSA systems. Tech. Rep. EtnoTeam. http://www.cee.etnoteam.it/prosa

GARG, P. AND SCACCHI, W. 1989. Ishys: Designing an intelligent software hypertext system. *IEEE Expert*.

GORLEN, K. E., ORLOW, S. M., AND PLEXICO, P. S. 1990. *Data Abstraction and Object-Oriented Programming in C+*. John Wiley & Sons, Inc., New York, NY.

HENNINGER, S. 1997. An evolutionary approach to constructing effective software reuse repositories. *ACM Trans. Softw. Eng. Methodol. 6*, 2, 111–140.

HOPKINS, T. AND PHILLIPS, C. 1988. *Numerical Methods in Practice: Using the NAG Library*. International Computer Science Series. Addison-Wesley, Reading, MA.

IBM. 1990. IBM AIX Version 3 for RISC System/6000: Command reference. T. J. Watson Research Center, IBM, Yorktown Heights, NY.

JARKE, M. 1993. Vision driven systems engineering. In *Proceedings of IFIP TC8/Wg8.4 Working Conference on Information System Development Process* (Como, Italy), N. Prakash, C. Rolland, and B. Pernici, Eds. North-Holland Publishing Co., Amsterdam, The Netherlands.

JOHNSON, R. 1997. Frameworks = (Components + Patterns). *Commun. ACM 40*, 10 (Oct.), 39–42.

KATZ, S., RICHTER, C. A., AND THE, K.-S. 1987. PARIS: A system for reusing partially interpreted schemas. In *Proceedings of the 9th International Conference on Software Engineering* (Monterey, CA, Mar. 30–Apr. 2), W. E. Riddle, Ed. IEEE Computer Society Press, Los Alamitos, CA, 377–385.

KLIR, G. J. AND FOLGER, T. A. 1988. *Fuzzy sets, Uncertainty, and Information*. Prentice-Hall, Inc., Upper Saddle River, NJ.

KOSKO, B. 1992. *Neural Networks and Fuzzy Systems: A Dynamical Systems Approach to Machine Intelligence*. Prentice-Hall, Inc., Upper Saddle River, NJ.

KRUEGER, C. W. 1992. Software reuse. *ACM Comput. Surv. 24*, 2 (June 1992), 131–183.

LEVAN, H. 1996. Charlie: System for retrieving and reusing C++ classes. In *Proceedings of TOOLS Europe '96* (Paris, France).

LILLIE, C. 1991. Now is the time for a national software repository. In *Proceedings of AIAA on Computing in Aerospace* (Baltimore, MD, Oct.).

MAAREK, Y. S., BERRY, D. M., AND KAISER, G. E. 1991. An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Softw. Eng. 17*, 8 (Aug. 1991), 800–813.

MAIDEN, N. 1991. Analogy as a paradigm for specification reuse. *Softw. Eng. J. 6*, 1 (Jan. 1991), 3–15.

MEYER, B. 1990. *Eiffel: The Libraries*. Prentice-Hall, Inc., Upper Saddle River, NJ.

MI, P. AND SCACCHI, W. 1990. A knowledge-based environment for modeling and simulating software engineering processes. *IEEE Trans. Knowl. Data Eng. 2*, 3 (Sept.), 283–294.

MILI, R., MILI, A., AND MITTERMEIR, R. T. 1997. Storing and retrieving software components: a refinement based system. *IEEE Trans. Softw. Eng. 23*, 7, 445–460.

MILI, A., MILI, R., AND MITTERMEIR, R. 1989. A survey of software reuse libraries. *Ann. Softw. Eng. 5*.

MOORMANN ZAREMSKI, A. AND WING, J. M. 1995. Signature matching: A tool for using software libraries. *ACM Trans. Softw. Eng. Methodol. 4*, 2 (Apr. 1995), 146–170.

ZAREMSKI, A. M. AND WING, J. M. 1997. Specification matching of software components. *ACM Trans. Softw. Eng. Methodol. 6*, 4, 333–369.

MOTRO, A. 1988. VAGUE: A user interface to relational databases that permits vague queries. *ACM Trans. Off. Inf. Syst. 6*, 3, 187–214.

OSTERTAG, E., HENDLER, J., DÍAZ, R. P., AND BRAUN, C. 1992. Computing similarity in a reuse library system: An AI-based approach. *ACM Trans. Softw. Eng. Methodol. 1*, 3 (July 1992), 205–228.

PAUL, S. AND PRAKASH, A. 1994. A framework for source code analysis using program patterns. *IEEE Trans. Softw. Eng. 20*, 6 (June), 463–475.

PFLEEGER, S. L. 1996. Measuring reuse: A cautionary tale. *IEEE Softw. 13*, 4 (July), 118–125.

PODGURSKI, A. AND PIERCE, L. 1993. Retrieving reusable software by sampling behavior. *ACM Trans. Softw. Eng. Methodol. 2*, 3 (July 1993), 286–303.

PRIETO-DÍAZ, R. 1991. Implementing faceted classification for software reuse. *Commun. ACM 34*, 5 (May), 88–97.

PRIETO-DÍAZ, R. 1993. Status report: Software reusability. *IEEE Softw. 10*, 3 (May), 61–66.

PRIETO-DÍAZ, R. AND FREEMAN, P. 1987. Classifying software for reusability. *IEEE Softw. 4*, 1 (Jan.), 6–16.

SALTON, G. 1971. Experiments in automatic thesaurus construction for information retrieval. In *Proceedings of the IFIP Congress Foundations of Information Processing* (Aug.). IFIP, Laxenburg, Austria.

SALTON, G., Ed. 1988. *Automatic Text Processing*. Addison-Wesley Series in Computer Science. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

SALTON, G. AND BUCKLEY, C. 1988. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manage. 24*, 5 (1988), 513–523.

SALTON, G., ALLAN, J., AND BUCKLEY, C. 1994. Automatic structuring and retrieval of large text files. *Commun. ACM 37*, 2 (Feb. 1994), 97–108.

SCHMIDT, D., FAYAD, M., AND JOHNSON, R. 1996. Software patterns. *Commun. ACM 3*, 10 (Oct.).

SEN, A. 1997. The role of opportunism in the software design reuse process. *IEEE Trans. Softw. Eng. 23*, 7, 418–436.

SPANOUDAKIS, G. AND CONSTANTOPOULOS, P. 1994. On evidential feature salience. In *Database and Expert Systems Applications, Proceedings of the 5th International Conference on Expert Systems Applications* (DEXA '94, Athens, Greece, Sept. 7–9), D. Karagiannis, Ed. Lecture Notes in Computer Science, vol. 856. Springer-Verlag, New York, 153–157.

TANG, Y. Y., YAN, C. D., AND SUEN, C. Y. 1994. Document processing for automatic knowledge acquisition. *IEEE Trans. Knowl. Data Eng. 6*, 1 (Feb.), 3–21.

TAYLOR, R. N., BELZ, F. C., CLARKE, L. A., OSTERWEIL, L., SELBY, R. W., WILEDEN, J. C., WOLF, A. L., AND YOUNG, M. 1988. Foundations for the Arcadia environment architecture. *SIGPLAN Not. 24*, 2 (Feb.), 1–13.

YU, C. 1975. A formal construction of term classes. *J. ACM 22*, 1.

Received: January 1997; revised: July 1997 and July 1998; accepted: December 1998