1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

A UNIFIED MODEL FOR SOFTWARE COMPONENT CLASSIFICATION

Yuen-Chang Sun and Chin-Laung Lei

Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan ROC
E-mail: sun@fractal.ee.ntu.edu.tw, lei@cc.ee.ntu.edu.tw

ABSTRACT

This paper introduces a unified software component classification model. It unifies four widely used retrieval methods, namely faceted methods, free-text methods, enumerative methods, and keyword-based methods, making them special cases. Users can choose and combine the most appropriate method or methods when searching for components so that the accuracy in component retrieval can be improved. The internal classification structure is described. The components in a repository can be retrieval with both query and browse mechanisms. The query algorithm incorporates techniques such as thesauri and degradation functions to maximize the coverage of query results. A simple component identification and version control scheme is also introduced. With this scheme, unique identifiers can be assigned to components without a cataloging service, and component corruptions can be detected easily.

1 INTRODUCTION

Software repositories play a central role in component-oriented software reuse. For a repository that cotains a none trivial collection of software components, the importance of an efficient, accurate and easy to use retrieval method cannot be over-emphasized. Various types of retrieval methods have been proposed, including those surveyed in [1, 2], and, since each of them has advantages and disadvantages, it is not uncommon for repositries to incorporate multiple methods simultaneously. The incorporated methods, however, are usually treated as parallel and independent facilities, increasing the complexity in repository management, and excluding the possibility of sharing classification information and resources between methods.

This paper introduces a classification model which unifies all the major types of retrieval methods. All the classification information are kept in a unified abstract data structure. The whole classification scheme is operated with one unified user interface, and communications between repositories are done with one unified protocol, no matter which types of retrieval methods are being used. With this unified model, repository management can be simplified, and classification information can be utilized more effectively. Furthermore, using a unified communication protocol makes it possible for repositories to cooperate: a query can be passed from one repository to another automatically when appropriate.

The proposed model only defines the abstract behavior of a repository. It is flexible in that a repository can choose to support a subset of the methods the model supports, and in the meantime the user interface and the communication protocol behave the same way. Also, the repository can choose any implementation that is suitable to the chosen retrieval methods so that system performance can be optimized, as long as the abstract requirements of the model are fulfilled.

Also introduced as part of the model is a component identification and version control scheme. Unlike schemes that are based on centralized cataloging services, the scheme proposed in this paper is based on content-derived naming techniques, and has the advantages that (1) a unique identifier can be derived purely from the contents of a component, eliminating the need for a networked or centralized service; (2) it is extremely difficult to produce a component that has a given identifier, protecting the contents of components from accidental or malicious alteration.

The following begins with a brief survey of existing retrieval methods. Then in Section 3 and Section 4 the basic structure of the proposed model is shown. Section 5 describes how components can be retrieved. Finally we conclude in Section 6.

2 RETRIEVAL METHODS

Before a retrieval method can work, the components in a repository must be arranged in a *classification structure* so that a component meeting certain criteria can be found efficiently. In the classification processa component may need to be assigned an *index* that describes its characteristics. Existing retrieval methods fall roughly into five categories, namely enumerative

methods, keyword-based methods, faceted methods, text-based methods, and specification-based methods. They are summarized in Table 1.

The classification structure of enumerative methods is a tree, like a directory tree found in a file system. Enumerative classifications are easy to understand and use because of their highly structured nature. Another advantage, which is missing in other methods, is that the user can traverse the whole classification structure, making them not only able to locate components without intimate knowledge about the vocabulary and structure of the repository, but also able to acquire this knowledge during the traversing process. One disadvantage of enumerative methods is inflexibility. Once the classification structure is determined, only one viewpoint is allowed. A common solution to this problem is providing two or more classifications simultaneously, but this increases maintenance cost.

Keyword-based methods, text-based methods and specification methods all have a linear classification structure; they differ in how they index components. In the keyword-based case, components are indexed with a number of keywords. Keyword-based systems are easy to implement and thus are widely used. The disadvantages are (1) the keywords must be manually assigned, a quite labor-intensive and time-consuming job; (2) the user must be familiar with the vocabulary of the repository, or the desired components may be missed.

The classification structure of faceted methods is similar to a relational database table. The columns are *facets*, and the cells are filled with *terms*. Each column identifies one attribute of components, like their functions, or the data structures they manipulate. Each row corresponds to a component, and the index of the component is then composed of the terms in the row. In contrast to enumerative methods, faceted methods are flexible in the viewpoint problem: the user can switch from one view to another at any time, as long as there are facets form the views. Furthermore, facets are assumed to be independent attributes of components, so modifying a facet does not affect other facets, making it easy to change the classification structure. Faceted methods also suffer from the difficulty in indexing components, like keyword-based methods. Besides, faceted methods are not suitable for collections that cover multiple application domains because two domains usually need two different facet sets.

With text-based methods, the text part of a component is used as its index. The text part usually includes the comments in the source files and the documents, and thus can be automatically extracted from the component, so the text-based approach is very low-cost. On the other hand, text-based methods are less effective for software component retrieval. The reasons are (1) false alarms happen frequently because the presence or absence of a term in a document may not mean its relevance or irrelevance to that document; (2) the component may under-documented or even undocumented, or the documentation quality is poor, especially when non-descriptive or non-standard terms are used. An empirical study [11] shows that it costs more time to locate components with text-based methods than with keyword-based or faceted methods.

3 COMPONENTS

3.1 Component ID

Components are identified by *component IDs*. A component ID is an MD5 [15, 16] code derived from the contents of the component. The MD5 code of a document is derived purely from the document contents by applying a one-way hashing function to map the contents to a code in a 128-bit code space. Due to the hugeness of the code space, it is extremely unlikely to have two distinct documents mapping to the same MD5 code. In case there are 10^{15} documents, the probability that any two of them being mapped to the same MD5 code is about 10^{-9}, which can be neglected for practical use.

A component is assigned a component ID once it is released. Since the ID is content-derived, a new component ID must be generated whenever the component is changed. One advantage of using MD5 codes as component IDs is that it is virtually certain that two components are identical if and only if their component IDs are identical, making MD5 codes more reliable in identifying components than traditional name-plus-version-number scheme, because name conflicts are more likely to happen than MD5 code conflicts. Of course, with a cataloging service ID conflicts can be totally avoided, but the content-derived scheme can works without such a service. An additional advantage is that corruption of component contents can be detected, no matter the corruption is caused incidentally or maliciously (e.g. infected by a virus), by checking if the contents and the ID matches.

3.2 Version Control

The component IDs of two distinct components are completely unrelated. In order to identify the relationship between related components, three more IDs, namely the *species ID*, the *family ID*, and the *parent ID*, plus a version number, are introduced as the version control scheme.

Two components should have the same fundamental functionality and purposes to be in the same species and to have the same species ID. In a species, components that work in different environments should belong to different families and have different family IDs. Environmental differences include the differences in software, hardware, programming language, and even culture (such as language, time format, number representation, currency, and so on). If one component is a modification of another, the latter is said to be the

category	index	structure	examples
enumerative	—	tree	RSL [3], IMSL [4]
keyword-based	keywords	flat	RSL [3], REUSE [5]
faceted	term tuple	rel. table	Prieto-Dìaz and Freeman [6, 7], Poulin [8]
text-based	text	flat	Practitioner [9], GURU [10]
spec.-based	specification	flat	Zaremski and Wing [12, 13], Mili *et al.* [14]

Table 1: Categories of retrieval methods.

parent of the former, and the parent ID of the former is the component ID of the latter. Finally, in a family a newer component should have a version number larger than that of an older component.

The rules for setting version information of a component under various circumstances are summarized in Table 2. Note how the first component in a species or a family is treated differently from its descendants. Also note that the "MD5 of contents" is not the component ID; this is explained in Section 3.3. With the version information the family tree of a species can be established. Although the version control scheme can work without any cataloging service, such a service is needed if the user is interested in constructing a family tree. Building such a service is a simple matter and is not covered in this research.

3.3 Component Structure

The data contained in a component are organized as *parts* and *fields*. A component has four parts, namely the *header*, the *subheader*, the *body*, and the *appendix*. Each part contains a number of fields. The parts of a component and the fields they can contain are summarized in Table 3. An entry in this table indicates the number of occurrences of a field allowed in a part.

The body is the most crucial part of a component. It contains fields that should never be modified once the component is released. If changes have to be made to those fields, a new component must be released. The *appendix* part contains information that may vary. The fields in the appendix can be translations of the fields in the body part (such as document and abstract) to a regional language, or referential information that can vary in the future (such as links), or the data dedicated to component classification (the index field). Information in the appendix can be altered whenever needed, and multiple variations of a component (with the same component ID) can coexist. The subheader contains version information.

The *code field* contains all the code needed for the component to be utilized. There can be binary code and/or source code. If binary code presents, it must be stored in the code field of the body part. If the source code presents and it is essential for using the component, it must also be placed in the body part, otherwise it should be stored in the code field of the appendix part. The code field of the body part is special in that it can be removed when needed. For example, the repository can choose not to expose the code field until the user has assessed the component (by studying the documentation) and decided to pay for it, if it is a commercial product.

The documentation is stored in the *document field*, and the *abstract field* should be briefed from the document. They are textual data, suitable for a text-based search. The *name*, *author*, and *company* fields identify the name, developer, and owning organization of the component, respectively. The *link fields* contain URLs pointing to locations related to the component. The *index field* contains the component index, which is described in Section 4.1. The *component time* and *appendix time* fields record the time the component is released and the time the current appendix is composed, respectively.

When a component is to be released, first the body and the appendix are constructed, then the fields in the subheader is assigned values according to Table 2. Finally the appendix ID is derived from the appendix part, the component ID is computed as the MD5 code of the concatenation of the subheader and the body, and the header part is then composed of the component ID and the appendix ID. If a specific variation of a component is needed, the repository can be searched with both the component ID and the appendix ID specified. Otherwise, the component ID alone can be used as the search key.

4 CLASSIFICATION STRUCTURE

4.1 Indices

An *index* is composed of a set of *terms*. A term can be accompanied by a *facet*, and such a term is called a *factor*. In their textual form, a term is merely a string, and a factor is a pair of strings separated by a "=" sign, the latter string being the term and the former string being its accompanying facet. Factors are allowed to share a facet. The order of terms is immaterial.

4.2 Domains and Classes

A *class* is a collection of components. Each component is assigned an index to describe it. The components in a class may have different indices, but they must have some common characteristics to be classified in one class. A component can reside in multiple classes.

	first in species	first in family	otherwise
species ID	MD5 of contents	species ID of parent	species ID of parent
family ID	MD5 of contents	MD5 of contents	family ID of parent
parent ID	0	component ID of parent	component ID of parent
version	arbitrary	arbitrary	> version of parent

Table 2: Rules for setting version information.

	header	subheader	body	appendix
component ID	1			
appendix ID	1			
species ID		1		
family ID		1		
parent ID		1		
version		1		
code			1*	$0 \sim 1$
name			1	$0 \sim 1$
abstract			1	$0 \sim 1$
document			1	$0 \sim 1$
author			$0 \sim n$	$0 \sim n$
company			$0 \sim n$	$0 \sim n$
component time			1	
link				$0 \sim n$
index				1
appendix time				1

Table 3: Parts and fields of a component.

A *domain* is a collection of classes. Each class is assigned an index that is composed of factors only. Furthermore, the indices of the classes in a domain share a common set of facets, and the indices must be distinct. The domain thus forms a traditional faceted classification structure, and its facet set is the facet set of the class indices. The facet set of domain may be empty, in which case at most one class with an empty index can be in that domain. A class must reside in exactly one domain.

Domains are organized as a tree hierarchy. Each domain in this hierarchy is assigned a name, and siblings must have distinct names. In addition to a name a domain is also assigned an index. The whole classification structure is demonstrated in Figure 1.

4.3 Other Issues

The introduced classification structure smoothly unifies all the common classification methods, making them special cases. More specifically, this structure is degenerated to

- an enumerative structure if all domains are unfaceted and all indices (of any kind) are empty;
- a text-based classification structure if there is only one unfaceted domain, all indices are empty, and component documents are used as the search target (this is explained in Section 5);
- a keyword-based classification structure if there is only one unfaceted domain, all domain indices and class indices are empty, and all component indices contain terms only;
- a faceted structure if there is only one faceted domain and all indices but class indices are empty.

In general, the classification structure of a repository can be any combination of these traditional structures. It is important to note here that the proposed model only defines the logical classification structure a repository should present to the client so that a client can communicate with different repositories using the same protocol and user interface. A repository implementation can choose any appropriate internal representation according to which combination of classification structures is supported.

Component indices, class indices and domain indices are independent to each other. When trying to match a component in a class in a domain with a query, all three indices are combined andcompared to the query. Having more than one index associated to a component is important to effective component classification. Component indices are supposed to describe the general functionality and purposes of components. On the other hand, domain indices and class indices are supposed to describe specific problem or application domains. If a general-purpose component can be utilized in several application or problem domains, it can be classified accordingly by placing it in multiple classes. This solves the problem that whether the computa-

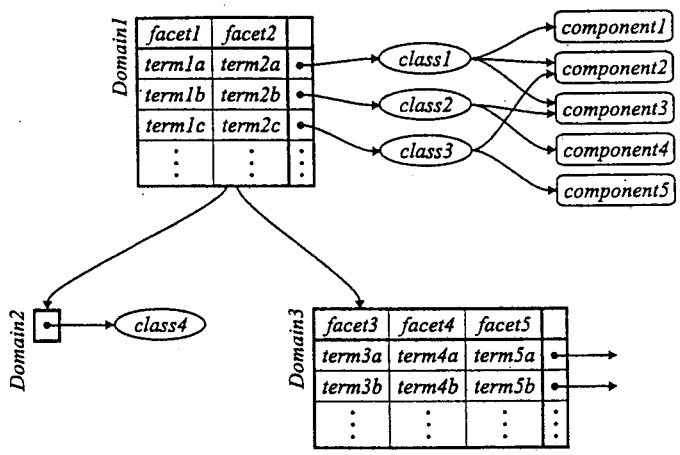

Figure 1: An example classification structure. *Domain1* is the root domain and has two facets. *Domain3* has three facets, and *Domain2* is unfaceted. Note that a component can be shared by multiple classes.

tional semantics, as in "this component finds the highest value in a list," or the application semantics, as in "this component finds the highest paid employee," should be described by the index of a component. The answer here is *both*: the former can be described by the component index, and the latter can be described by the class index and the domain index.

5 COMPONENT RETRIEVAL

5.1 Syntax of Query Statements

The syntax of query statements is listed in Figure 2. In QueryStat a "+" stands for a union operation, a "#" an intersection, and a "−" a difference. In QueryItem a "," stands for an AND operation. The type of a Const and the kind of Op allowed depend on the type of Facet and the Func applied, if they present.

When a Const is accompanied by a Facet (possibly with a Func applied) and an Op, that constant is compared to the specified facet. Otherwise, that constant is compared to all the facets and terms of the same type in the component.

5.2 Thesauri

When using string-based methods such as keyword-based and faceted methods, a major problem is the difference between the user vocabulary and the repository vocabulary. For example, a window may be called a "form" in some field, and, though not identical, it has a high correlation with a "dialog box." Listing all equivalent or similar terms in an index is out of question. The best solution to this problem is a thesaurus mechanism, with which the correlation between two strings can be found. Such a thesaurus mechanism can define the correlation between general terms, general terms and special terms such as abbreviations and acronyms, or terms from different languages such as Chinese and English.

A thesaurus is defined as a function $\theta(w_1, w_2) \mapsto [0..1]$, where θ is the thesaurus and w_1, w_2 are strings. The larger the function value, the higher the correlation between the strings. A correlation 1 means the two strings are equivalent, while a 0 means irrelevance. $\theta(w_1, w_2)$ can also be denoted as $|w_1, w_2|$. In practice several thesauri may be used simultaneously, but it is feasible to view them as a single thesaurus.

A thesaurus can be implemented in many ways. The simplest way is adopting an existing thesaurus (such as ones used in some word processors) and restrict the function value to either 0 or 1, hence a binary thesaurus. More complicated techniques such as conceptual graphs introduced in [6] can also be used. Besides, language specific procedures such as suffix elimination can also be incorporated. Establishing a thesaurus is not easy, and automatic or adaptive methods can be of great help. This paper does not cover this topic.

5.3 Relevance Evaluation

Executing a query involves comparing the query statement to the indices of components in the repository to find the correlations, and then present to the user the most highly relevant ones for further assessment. The index of a component is the union of the component index, class index and domain index of that component. More specifically, if a component has component index d_c, class index d_l, and domain index d_a, then its index used in a query session is $d_c \cup d_l \cup d_a$. The class index and domain index of a component is the indices of the class and domain, respectively, which that component is residing in. If a component resides in multiple classes, it is considered as multiple instances and will have multiple indices during a query.

Now suppose an index $d = (r_1, r_2, \ldots, r_n)$ is compared to a query item $q = (s_1, s_2, \ldots, s_m)$, where r_i is a term or a factor, and s_i is a query element (QueryElem in 2). Then the correlation between q and d, also called

QueryStat	⇒	QueryItem | QueryStat' "+" QueryStat' |
		QueryStat' "#" QueryStat' | QueryStat' "−" QueryStat'
QueryStat'	⇒	"(" QueryStat ")"
QueryItem	⇒	QueryElem "," QueryItem
QueryElem	⇒	Const | Facet Op Const | Func(Facet) Op Const
Op	⇒	"=", "≠", ">", "<", "≥", "≤"

Figure 2: Syntax of Query Statements

the *relevance* of d against q, is defined as

$$|q,d| = \prod_{i=1}^{m} \left(\max_{j=1}^{n} |s_i, r_j| \right).$$

$|s_i, r_j|$ is evaluated according to what kinds of elements are involved:

- $|t_1, t_2|$, where t_1 and t_2 are strings, is evaluated by consulting the thesaurus;
- $|f_1 = t_1, t_2| = |t_2, f_1 = t_1| = |t_1, t_2|$ for string-type facet f_1;
- $|f_1 = t_1, f_2 = t_2| = \sqrt{|f_1, f_2| \cdot |t_1, t_2|}$ for string-type facets f_1 and f_2;
- $|f_1 \text{ op } t_1, f_2 = t_2| = \sqrt{(t_1 \text{ op } t_2)\,|f_1, f_2|}$ for facets f_1 and f_2 of the same non-string type that is suitable for op;
- $|\mathrm{F}(f_1) \text{ op } t_1, f_2 = t_2| = \sqrt{(\mathrm{F}(t_1) \text{ op } t_2)\,|f_1, f_2|}$;
- otherwise $|s_i, r_j| = 0$.

The value range of the expression "a op b" depends on op. Usually it is {0, 1}, but for the text type, fraction values are possible if partial matching techniques are used. After the relevance of each query items are found, the relevance of the whole combined index can be found:

- $|q, d|$ is evaluated as the above if q is a query item;
- if $q = q_1 + q_2$, where q_1 and q_2 are sub-queries of q, then $|q, d| = \max(|q_1, d|, |q_2, d|)$;
- if $q = q_1 \# q_2$, then $|q, d| = \min(|q_1, d|, |q_2, d|)$;
- if $q = q_1 - q_2$, then $|q, d| = |q_1, d| - |q_2, d|$.

Note that during the evaluation process, due to the subtraction operation there can appear values out of the [0..1] range. Out-of-range values should be truncated after the evaluation, but allowing them to appear during evaluation has its positive effects. For example, consider a query $q = q_1 - (q_2 - q_3)$, where q_1, q_2 and q_3 are query items. If a combined index d does not match q_1 and q_2, then no matter d matches q_3 or not, d will not match q if the traditional binary set operation (which is equivalent to truncating out-of-range values as soon as they appear during an evaluation process) is used. On the other hand, $|q, d|$ depends on $|q_3, d|$ if truncation only takes place after evaluation. Since $|q, d|$ is an evaluation of analog correlation, not simple binary "contained-in" relation, our approach seems more practical. Of course, the traditional semantics can also be used if the user prefers it.

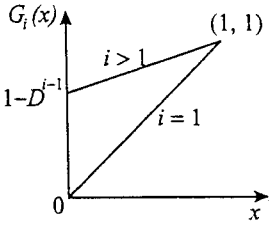

Figure 3: Degradation Functions

5.4 Degradation Functions

One way [6] of broadening a query is dropping factors and terms one by one from the tail of the original query statement and reissuing the shortened queries after confirmed by the user. In this paper an advanced broadening method is introduced. The original algorithm for finding the correlation between a query item and an index is changed to

$$|q,d| = \prod_{i=1}^{m} G_i \left(\max_{j=1}^{n} |s_i, r_j| \right)$$

where G_i is the ith order degradation function defined as

$$G_i(x) = 1 - D^{i-1}(1 - x)$$

where $0 < D \leq 1$ is the *degradation constant*. See Figure 3, this function has the following characteristics:

- $G_1(x) = x$ for all x, that is, the first element in the query is not affected;
- $G_i(1) = 1$ for all i, that is, an exact match is not affected;
- $G_i(0) > 0$ for all $i > 1$, and the larger the value i, the larger the function value, that is, when elements in a query item do not match the index exactly, an element arranged closer to the tail has less effect over the whole relevance value than an element arranged farther to the tail.

This method has the following advantages compared to the method proposed in [6]:

- The query has to be issued only once, rather than multiple times as is done in [6].
- It is automatically done without user intervention.
- The relevance of a component after broadening is proportional to its relevance before broadening, so a more matching component receives a higher relevance value even when broadening is performed.

The last point needs further explanation. Suppose the relevance threshold is 0.25. Since merge sort is close to quick sort in efficiency, let |quick sort, merge sort| be 0.2. Without degradation functions applied the relevance values of bubble sort and merge sort are 0 and 0.2, respectively, so both are dismissed. With degradation functions applied, their relevance values become 0.3 and 0.44, respectively. Note that although both are now included in the result set, the merge sort component has a higher relevance, thus the system can arrange it closer to the top in the list of located components so that it can more easily be retrieved by the user. On the other hand, the method in [6] will give a relevance value of 1 to both of them since the second factor is dropped, and the system will not be able to tell which one is more matching.

5.5 Browsing

In addition to the query mechanism, components can also be retrieved with a browse mechanism. The browse starts with the root domain. The user can traverse the tree of domains to locate the desired domain. Then the located domain can be surveyed and classes in the domain can be selected. Finally components can be retrieved from the classes. This way the user can enumerate through each component in a repository. The browse mechanism also offers the user an opportunity to get familiar with the vocabulary and classification structure of the repository.

A domain is a faceted classification structure, which has a natural representation of a table. Under certain circumstances, however, it is more convenient for users to represent the faceted structure as a tree. A retrieval system implementing the proposed model should provide both representations, and the user should be allowed to switch representation with ease.

Representing a domain A with facets $f_1, \ldots, f_n$ as a tree is a straightforward task. The domain can first be expanded with f_1 by performing appropriate relational operations against the domain table to yield several sub-domains with f_1 excluded from their facet sets. The expanded domain A becomes the tree root, and the sub-domains become its child nodes. Those sub-domains can then be expanded with f_2. This process continues until facets are exhausted, as illustrated in Figure 4.

Each node in the tree stands for a subset of the whole domain. More specifically, if a node is derived from A by the operation series $(f_1 = t_1, \ldots, f_m = t_m)$, then it contains all the classes having class indices of the form $(f_1 = t_1, \ldots, f_m = t_m, f_{m+1} = *, \ldots, f_n = *)$, where "$*$" stands for immaterial. In other words, the class set of a node is the union of the classes sets of its child nodes, and the root node stands for all the classes in the domain. For example, the node *edit* in Figure 4 stands for all the classes that are assigned the term "edit" to their "task" facet, that is, classes C_1 and C_2.

Facets can be expanded in any order. For example,

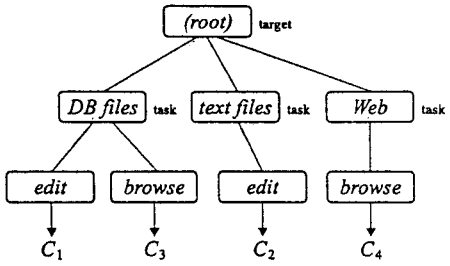

Figure 5: The domain in Figure 4 is expanded in another facet order.

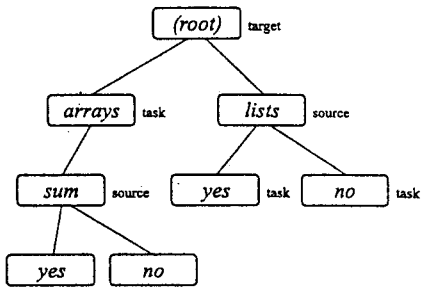

Figure 6: Expanding different facets at the same level. The domain is partially expanded.

in Figure 5 the same domain as shown in Figure 4 is expanded with "target" first. Nodes at the same level may be expanded with different facets, as shown in Figure 6.

Some repository manipulation tasks can also be done with this tree representation. Locating a class (and the components in it) can be done by expanding part of the tree, and assigning or changing the class index of a component can be as simple as moving the component around the tree structure. Certain operations, especially those involving facets, are better performed with the table representation, though.

6 CONCLUSION

This paper introduces a software component classification model that unifies four major retrieval methods. Repositories incorporating this unified model can be accessed using a common protocol and user interface, and cooperation between repositories can be encouraged. When searching for components, a user can choose the access method that is most appropriate to the situation, reducing the possibility that the desired components cannot be found. The user can also use a mixture of methods, increasing the expressiveness of a query statement, and thus improving the accuracy of query results. A query can be understood and performed simultaneously by multiple repositories so that more components can be found.

Different repositories can choose to support different subsets of the four retrieval methods, and different internal data structures can be chosen to optimize per-

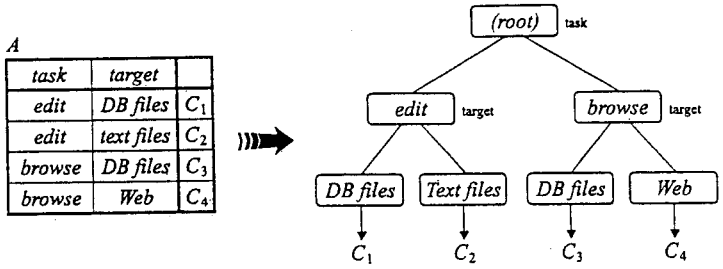

Figure 4: Representing a Domain as a Tree

formance. The query and browse mechanisms are designed so that they can work with various combinations of retrieval methods.

In addition to component classification, a simple yet effective version control method is also introduced. With this method, components can be assigned quasi-unique identifiers without a centralized catalog service, and relationship between versions of a component can be easily determined. Because of the encoding scheme used, it is extremely difficult to break the integrity of a component.

REFERENCES

[1] H. Mili, F. Mili and A. Mili, "Reusing Software: Issues and Research Directions," *IEEE Trans. Software Engineering*, vol. 21, no. 6, pp. 528–562, 1995.

[2] S. Henninger, "An Evolutionary Approach to Constructing Effective Software Reuse Repositories," *ACM Trans. Software Engineering and Methodology*, vol. 6, no. 2, pp. 111–140, 1997.

[3] B. A. Burton, R. W. Aragon, S. A. Bailey, K. D. Koehler and L. A. Mayes, "The Reusable Software Library," *IEEE Software*, vol. 4, no. 7, pp. 25–33, 1987.

[4] *IMSL Math/Library User's Manual*, Houston, Texas, 1987.

[5] S. P. Arnold and S. L. Stepoway, "The Reuse System: Cataloging and Retrieval of Reusable Software," *Proceedings of COMPCON S'87*, pp. 376–379, 1987.

[6] R. Prieto-Dìaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software*, vol. 4, no. 1, pp. 6–16, 1987.

[7] R. Prieto-Dìaz, "Implementing Faceted Classification for Software Reuse," *Communications of the ACM*, vol. 4, no. 5, pp. 88–97, 1991.

[8] J. S. Poulin, "Populating Software Repositories: Incentives and Domain-Specific Software," *Journal of Systems Software*, vol. 30, pp. 187–199, 1995.

[9] H. Mili, R. Radai, W. Wang, K. Strickland, C. Boldyreff, L. Olsen, J. Witt, J. Heger, W. Scherr and P. Elzer, "Practitioner and SoftClass: A Comparative Study of Two Software Research Projects," *Journal of Systems Software*, vol. 25, pp. 147–170, 1994.

[10] Y. S. Maarek, D. M. Berry and G. E. Kaiser, "An Information Retrieval Approach for Automatically Constructing Software Libraries," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 800–813, 1991.

[11] W. B. Frakes and T. P. Pole, "An Empirical Study of Representation Methods for Reusable Software Components," *IEEE Trans. Software Engineering*, vol. 20, no. 8, pp. 617–630, 1994.

[12] A. M. Zaremski and J. M. Wing, "Signature Matching: A Tool for Using Software Libraries," *ACM Trans. Software Engineering and Methodology*, vol. 4, no. 2, pp. 146–170, 1995.

[13] A. M. Zaremski and J. M. Wing, "Specification Matching of Software Components," *ACM Trans. Software Engineering and Methodology*, vol. 6, no. 4, pp. 333–369, 1997.

[14] R. Mili, A. Mili and R. T. Mittermeir, "Storing and Retrieving Software Components: A Refinement Based System," *IEEE Trans. Software Engineering*, vol. 23, no. 7, pp. 445–460, 1997.

[15] R. L. Rivest, *The MD5 Message-Digest Algorithm*, RFC 1321, Network Working Group, April 1992.

[16] J. K. Hollingsworth and E. L. Miller, "Using Content-Derived Names for Configuration Management," *ACM Symposium on Software Reusability*, pp. 104–109, MA, USA, April 1997.