全分散式可重用軟體元件庫系統之設計
The Design of a Fully Distributed Reusable Component Repositories System

孫運璋 Yuen-Chang Sun
高銘麟 Ming-Lin Kao
雷欽隆 Chin-Laung Lei

國立臺灣大學電機系
Department of Electrical Engineering
National Taiwan University
{sun | kml | lei}@fractal.ee.ntu.edu.tw

摘要

軟體再利用在增進軟體生產力和品質上有極高的潛力，但由於種種原因卻未得業界充分利用。我們建立了一個全分散式的軟體元件庫系統，並提出諸如降階函數與元件遷移史等巧，部分解決了這些問題。

關鍵字：軟體元件庫，多面相分類，分散式系統，軟體再利用

Abstract

Software reuse is the most promising technology for improving software productivity and quality. Unfortunately, due to various reasons it has not been widely adopted. As a partial solution, a fully distributed software component repositories system is developed. Also proposed are techniques such as degradation functions and component migration history.

Keywords: component repositories, faceted classification, distributed systems, software reuse

1 Introduction

While software productivity has been rising steadily over the past decades, the gap between the demands placed on the software industry and what the industry can deliver is not closed. After several decades of intensive research in artificial intelligence and software engineering, few alternatives remain, and software reuse is one of the few approaches that promise to offer the productivity and quality gains the software industry needs.

Simply speaking, software reuse is constructing new software systems using existing software artifacts. These artifacts can be anything ranging from code fragments to environmental knowledge, and both the products of previous software projects and the processes deployed to produce them can be reused, hence a wide spectrum of reuse approaches [1, 2]. Customarily, these approaches are divided into two categories: *generative* and *compositional* [3]. This paper is focused on the compositional approach.

The compositional methods involve reusable software components that are integrated into the target system. There are three typical steps in this process: (1) A description about the desired component is specified. (2) A search against a component repository is performed, based on the description given. (3) The retrieved components, if any, are integrated into the target system, either as is (*black-box reuse*) or adapted (*white-box reuse*).

Software reuse is like a savings account: the more you put in, the more interest you can collect [4]. In order to make software reuse cost-effective, a large collection of reusable components is a necessity. Until recently, most research efforts in the software reuse field have been put upon the development of a large general-purpose mega-library that is owned by and used within an organization and does not communicate with other mega-libraries. This type of software repositories is conceptually simple and easy to implement. Managing and populating such a mega-library with reusable components, however, has its difficulties. First, the cost of designing, populating and maintaining a large general-purpose repository is usually higher than what an organization can afford. Second, the time for finding and adapting reusable components is often more than what a programmer can spend. In other words, it could take more to reuse software than to construct it from scratch, both for organizations and for programmers. Besides, this approach can not exploit the full potential of software reuse: while a component would (hopefully) not be invented twice within an organization, different components with the same functionality could be constructed in all organizations.

One solution to the above problems is to introduce a decentralized architecture, in which software repositories are interconnected with a network (more specifically, the Internet) and can thus share software resources [5, 6, 7, 8]. This approach has the following advantages. First, because a much larger number of components than what one repository can offer are available to developers, it is more likely an adequate component can be found to fulfill a requirement. Second, it has been pointed out [4] that the more often a component is used, the more hidden defects in it can be found and fixed. By allowing users other than the owner to access a component, its quality and reliability can be enhanced by being reused frequently. Third, since for one requirement there can be more than one competing choice, components of less quality will be less reused, and will finally be either improved or discarded. This way the evolution of software components can be sped up.

Despite the above advantages, the decentralized approach can solve reuse problems only to some extent. In this paper we go one step further by proposing a fully distributed architecture. In this architecture components are distributed (with possible replications) into many small repositories (called sites in this paper). Each site is kept as small as possible and contains components from only one application or problem domain. It can be owned and managed by an organization, a division, a group, or even an individual programmer, and each repository can has its own component classification scheme, while resource sharing among repositories is still enabled. In fact, in our design principles individual programmers or small groups are encouraged to have their own sites. The goals are to make repository configuration more flexible and to make software resource sharing more fine-grain.

At a first glance our approach is of little difference from the decentralized one; it is just that repositories are down-sized and further distributed. By taking a closer looking at the differences, however, three main characteristics in our approach can be revealed:

- *User-orientation.* In order to encourage end-users to have their own repositories, site management must be made as simple as possible. In our system a browse tool with a user-friendly graphical interface is introduced for both navigating through and manipulating components in a repository. Browsing hierarchy can be dynamically and intuitively built over the classification structure without additional effort. Components can be classified by less-experienced users with simple mouse actions. This way the managerial cost of repositories can be reduced and amortized over users.
- *Localization.* Because of the diversity among individuals, organizations, geographical areas, and so on, site owners should be allowed and encouraged to have different classification schemes in their sites. For example, a collection of components for building Chinese applications should be classified differently from a collection for building applications in other languages, because in a Chinese repository there must be additional components that can process double-byte character sets. Besides, even with the same collection the user can still benefit from having a private classification scheme, because component searching can be made more efficient for the user's familiarity with the site structure. In order to deal with the diversity in classification schemes, extended thesauri and multi-facet techniques are introduced in our system.
- *Component circulation.* Since the granularity of software resource sharing is finer in our approach than the decentralized one, it is much more frequent for components to be duplicated or moved from site to site. Site owners are allowed and encouraged to download components from other sites to their own sites for future reuse, and uploading is also possible. This way overall degree of software reuse can be further increased. Because component circulation is a routine practice, it is important to provide the user with the ability to track a component to its past locations. In our system a mechanism called *component migration history* is introduced for this task.

The classification method used in our system is an extension to the multi-facet method. Although this method has been severely criticized in, e.g., [3, 9, 10], we still decide to adopt it for the following reasons. First, this method is conceptually simple and easy to implement, and it operates efficiently during both the classification and retrieval process. Second, a browsing hierarchy can be naturally built upon a faceted classification structure, and one faceted structure can be represented dynamically as different hierarchies, depending on how the user wants to view the collection (see Section 6). Maarek *et al.* [11] proposed an off-line method to construct a hierarchy automatically from keywords, but there seems no intuitive way to make it on-line. Chou and Yang [12] proposed an on-line method, but the browsing hierarchy built is static. Third, since our system is intended to be managed at the end-user level, and since user-friendly tools are developed to facilitate component indexing, the up-front cost of populating repositories with components can be amortized and reduced. Also note that the original multi-facet method has been extensively enhanced in our research.

In the remaining of this paper we begin with an overview of the Uranus system in Section 2. In Section 3 our extended multi-facet classification method is shown. The migration history technique is introduced in Section 4. Then in Section 5 and 6 the internal mechanisms of the two main subsystems, the *query tool* and the *browse tool*, are described, and their implementation is briefed in Section 7. Finally we conclude in Section 8.

2 System Overview

The structure of the Uranus system is illustrated in Figure 1. The *library* is a collection of components without any classification structure. In order to distinguish components from each other, they are tagged with system-generated *globally unique identifiers* (GUIDs), which are guaranteed to be unique in both spatial and temporal senses. The classification information is stored in the *view*. Both the library and the view are manipulated by the *library server*. Querying, browsing and other housekeeping tasks can be performed with several tools, which form the client side of the system. When doing a query the *thesaurus server* and its associated *thesauri* are used to deal with synonyms.

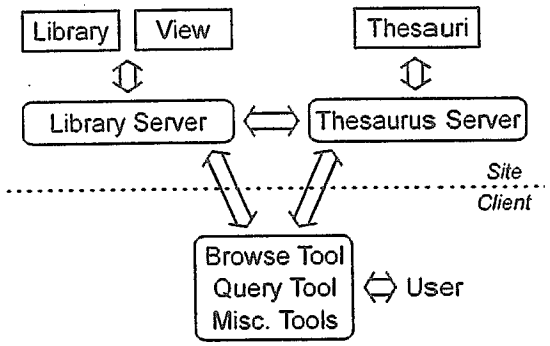

Figure 1: System structure overview.

Clients and servers can be arranged and secured in various configurations. For example, in the case of a personal site the client and the two servers usually all run on the same machine. In an organization, on the other hand, a central site can be established and shared by employees. This site can be secured so that outsiders can not access it. In addition to this central site, each employee can still has a private site. Finally, public sites can be established and shared by developers all over the community. Thesaurus servers can be arranged in a different configuration. For example, it is possible that several personal sites, each having its own library, view and library server, share the same thesaurus server and thesauri.

A user can perform a search (querying or browsing) against any site, as long as the user has appropriate access privilege to that site. Once the desired components are found, in addition to being reused, they can also be included in the private site of the user. When adding a component to a site, the site owner has full freedom in deciding where in the classification structure should that component be placed, without caring the classification structure of the site from which the component comes.

The query tool and the browse tool are used under different circumstances. When looking for components the user can choose to express the requirements as a query and issue the query with the query tool to one or more sites. On the other hand, it has been pointed out [11, 13] that having a browsing mechanism, which views the component collection as a hierarchy instead of a linear structure, is important for users to navigate through a repository lucidly. Thus, both tools are provided in our system. Details about choosing between them are described in Section 7.

3 Classification of Components

The component classification method used in our system is based on the faceted method proposed in [14, 15]. The following description, however, does not entirely follow their terminology and notions.

A *facet* is an attribute that all the involved components share. A *term* is a value assigned to a facet. A facet and its

associated term together form a *factor*. A list of factors form a *descriptor*. For example, (task = print, target = file) is a descriptor with two factors "task = print" and "target = file," in which "task" and "target" are facets and "print" and "file" are terms. The property of a component, including its functional and environmental information, can be described by one or more descriptors. Components described by the same descriptor form a *class*. Note that since a component can be described by more than one descriptor, the relationship between classes and components is not one-many but many-many.

In a view, all the components share the same facet set, and the facets in this facet set are ordered by their importance. For example, with the facet set (function, object, medium) the descriptor (function = sum, objects = values, medium = array) can be used to describe a component that sums up values in an array. If that component can also sum up values in a linked list, it can be described by an additional descriptor (function = sum, objects = values, medium = linked list). The set of all the descriptors describing a component is called the *index* of that component, and the action of assigning descriptors to a component is called *indexing*.

A query has the same form of a descriptor, except that arbitrary symbols (words or phrases) can be used in place of facets or terms, and there can be any number of factors. For example, (task = sum, objects = numbers) is a valid query that can be performed against a view with facet set (function, objects, medium), and it is likely that the descriptor (function = sum, objects = values, medium = array) can match this query.

Since queries and descriptors can have different vocabularies, thesauri are used to measure the correlation between symbols in them. A thesaurus Θ is a function that maps a pair of symbols to the interval [0,1]. The correlation between two symbols w_1 and w_2, denoted by $\|w_1, w_2\|_\Theta$, indicates how closely the two symbols are related: the larger the value, the higher the correlation. For simplicity we make thesaurus a symmetric function, that is, $\|w_1, w_2\|_\Theta = \|w_2, w_1\|_\Theta$ for any w_1 and w_2. Furthermore, we define $\|w, w\|_\Theta = 1$ for any w. A special symbol "*" is introduced as a wildcard symbol, that is, $\|*, w\|_\Theta = \|w, *\|_\Theta = 1$ for any w.

A missing factor in a query is treated as if the missing factor has a wildcard term, so the query $(f_1 = t_1)$ can be treated as $(f_1 = t_1, f_2 = *)$ and can match exactly to the descriptor $(f_1 = t_1, f_2 = t_2)$. With the thesaurus function defined, correlation between a query and a descriptor can also be defined, and the relevance of a class can then be defined as the correlation between its associated descriptor and the query. Finally the relevance of a component is defined as the largest relevance of its associated classes. During a query session the relevance values of all the components in the view are computed, and highly relevant components are listed by their order of relevance to form the query results. Details about the computation of correlation between queries and descriptors are in Section 5.

4 Migration History

Since component circulation is encouraged, it is helpful if the user has the ability to know the past locations of a component. As mentioned above, even in the same site a component can reside in several classes, so we define the component migration history H_σ^c of component σ in class c as an ordered list of classes $(c_1, c_2, ..., c_n)$, where c_1 through c_n denote distinct classes. Such a migration history indicates that before arriving at class c the component σ has been residing in c_n, c_{n-1}, ..., c_2 and c_1, in that order. They are called the *source classes* of σ, and c_n, in which σ is first introduced, is called the *origin class* of σ. These classes may be on different sites. In order to distinguish classes from each other, they are tagged with GUIDs, as in the case of components.

When a component is circulated around classes and sites, its migration history is updated accordingly to reflect the change in its location. If a component σ is copied or moved from a class c to another class c', the migration history $H_\sigma^{c'}$ associated to the newly added entry in c' is defined as $c \oplus H_\sigma^c = (c, c_1, ..., c_{i-1}, c_{i-1}, ..., c_n)$ if $c = c_i$ for some i, or $(c, c_1, ..., c_n)$ otherwise. That is, c is appended to the head of the old migration history to form the new migration history, and the duplicated class is removed if there is one. In case σ is copied to c', H_σ^c remains the same. In case σ is moved to c', H_σ^c is replaced with $c' \oplus H_\sigma^c$. Either σ is moved to another class or it is deleted from c, its migration history in c is retained.

Since a given component and components in one of its source classes have once been indexed the same way, it is likely that they are closely related to each other. This is especially true when the source class is the very origin class, because the owner of the site where the origin class resides in may be devoted to producing components of the same kind as the given one.

Migration history can be manually traced during retrieval by issuing appropriate commands. In this case, the system lists all the source classes, and the user can then open their corresponding sites with the browse tool to find more related components. Migration history tracing can also be done by the query tool automatically. Automatic tracing is explained in Section 5. In both cases, the classes found through migration history tracing are called the *related classes* of the component corresponding to the traced migration history (or the class containing that component). The components in a related class are called the *related components*.

The migration history mechanism can be used as "see also" links attached to components and classes. This way, it gives our system part of the functionality a hypertext system can offer.

5 Querying

In this section the query algorithm used by the query tool is discussed in detail. The discussion starts with the trivial case that the query and the view share a facet set. Then the degradation function technique is introduced to deal with more complex cases. Finally the general case and automatic migration history tracing are discussed.

5.1 The Trivial Case

If a query q and a descriptor d have the same facet set, that is, $q = (f_1 = t_1, ..., f_n = t_n)$ and $d = (f_1 = t_1', ..., f_n = t_n')$, the correlation between them, denoted by $\|q, d\|$, is defined as $\prod \Theta(t_i, t_i')$. Note that the order of facets is assumed to be irrelevant in this case: if the query is given as, say, $(f_2 = t_2, f_1 = t_1, ..., f_n = t_n)$, its factors will be reordered by the system before comparing it to the descriptors. Also note that a factor can be missing from the query, as mentioned in Section 3.

5.2 Degradation Functions

In certain cases the order of factors in a query is irrelevant, and the system can freely reorder the factors in a query to accommodate the descriptors, as in the trivial case above. In general, however, one query factor may take precedence over another. For example, while (action = sort, object = array) and (object = array, action = sort) are equivalent most of the time, the order of factors in the query (action = sort, object =

array, algorithm = quick sort) might be relevant because usually users care more about functionality than mechanism. If no component can match the query exactly, a second best choice that matches, say, (algorithm = bubble sort) might be satisfactory. In the Prieto-Diaz system [14] the system can expand the query by dropping factors one at a time from the tail with explicit user requests. In the case given, a first-time expansion yields the query (action = sort, object = array), and a bubble sort component can be found accordingly. In the Uranus system another mechanism called *degradation functions* is introduced to improve query expansion. When the query $(f_1 = t_1, f_2 = t_2, \ldots, f_n = t_n)$ is matched to a descriptor $(f_1 = t_1', f_2 = t_2', \ldots, f_n = t_n')$, in which the factors might have been reordered to accommodate the query, the function $\prod G_i(\|t_i, t_i'\|)$ is used to measure the correlation between them. The function G_i is the ith degradation function, defined as $G_i(x) = 1 - D^{i-1}(1-x)$ where the constant $0 < D \le 1$ is the *degradation coefficient*. The degradation function is a linear mapping from [0, 1] to [1 − D^{i-1}, 1], as illustrated in Figure 2. When $i = 1$, it is equivalent to the identity function. As i is increased, the slope of the function is decreased, reducing the importance of the corresponding query factor. This agrees with the intuition that a factor typed earlier is more important than one typed later.

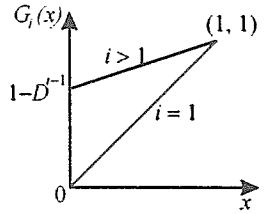

Figure 2: Degradation Functions

Broadening the query this way has two advantages. First, the expansion of queries is done automatically; the user does not have to explicitly issue an expansion command. Second, the components retrieved after the expansion can still be ordered by their relevance. In the above case, the system will put the descriptor (action = sort, object = array, algorithm = merge sort) above (action = sort, object = array, algorithm = bubble sort) because the performance characteristics of quick sort is closer to merge sort than bubble sort. The Prieto-Diaz method [14] does not have this property because the relevance information is stripped away with dropped factors.

5.3 The General Case

In most cases the facet set of the query differs from the facet set of the view, and thesauri must be used to translate between them. We first define the correlation between two factors u = "$f = t$" and u' = "$f' = t'$" as

$$\|u, u'\| = \|f, f'\| \cdot \|t, t'\|.$$

Then the correlation between the query $q = (u_1, u_2, \ldots, u_n)$ and the descriptor $d = (v_1, v_2, \ldots, v_m)$, where u_i and v_j are factors, is defined as

$$\|q, d\| = \prod_{i=1}^{n} G_i\left(\max_{1 \le j \le m} \|u_i, v_j\|\right).$$

Note that this is not a symmetric function; $\|q, d\| \neq \|d, q\|$ in general.

The above functions are not well-defined because the thesaurus used for measuring the correlation between symbols is not specified. In fact, because of the client/server nature of our system, at most three thesauri are necessary for this purpose. First, a *local thesaurus* Θ_L is used by the client to find synonyms at the client side. Then, a *foreign thesaurus* Θ_F is used, also by the client, to translate symbols to remote vocabulary. Finally, a *remote thesaurus* Θ_R is used by the library server to deal with synonyms in the site being queried against. The correlation between two symbols w_1 and w_2, with respect to these three thesauri, is thus defined as

$$\|w_1, w_2\| = \max_{v_1, v_2}(\|w_1, v_1\|_{\Theta_L} \|v_1, v_2\|_{\Theta_F} \|v_2, w_2\|_{\Theta_R}),$$

where v_1 and v_2 are arbitrary symbols.

Usually the local thesaurus and the remote thesaurus are essential for making a query. On the other hand, the foreign thesaurus is optional. For example, the foreign thesaurus is usually unnecessary when the client and the server are using the same language, say Chinese. Only when the server is using a foreign language, or when the remote vocabulary is very different from the local one, is the foreign thesaurus necessary. The user can assign each site a foreign thesaurus, and whenever a query is made against that site the assigned foreign thesaurus is used. If no foreign thesaurus is assigned, the correlation function degenerates to

$$\|w_1, w_2\| = \max_{v}(\|w_1, v\|_{\Theta_L} \cdot \|v, w_2\|_{\Theta_R}).$$

5.4 Automatic Migration History Tracing

At this point we must be aware again that a descriptor is associated to a class, not to a component. Without other clue components in a class are regarded as of equal relevance and will be (or not be) retrieved as a whole during a query session. When a class is found during retrieval, the migration history lists of its components are used to find related classes. The relevance of a related class is a function of the number of the occurrences of the class in the migration history lists. Suppose the history lists are $h_i = (c_1^i, c_2^i, \ldots, c_{m_i}^i)$, $1 \le i \le n$, the relevance value of a related class c is

$$\alpha \cdot \sqrt{\sum_{i=1}^{n} \frac{u_i(c)}{n}},$$

where

$$u_i(c) = \begin{cases} 1 & \text{if } c \in \{c_1^i, \ldots, c_{m_i}^i\}, \\ 0 & \text{otherwise,} \end{cases}$$

and $0 < \alpha < 1$ is a constant. This value is then multiplied to the relevance values of the components and classes retrieved from the related class. Since a class can appear in a migration history at most once, this value is always less than one.

For example, suppose in class c half of the components have been in another class c' before arriving in c, then c' will be found, provided c has been found, with relevance $\alpha\sqrt{1/2} \approx 0.707\alpha$ times the relevance of c. If components in c are retrieved, it is thus quite likely that components in c' will also be retrieved. For another example, consider the case when components in a class c are evenly moved to two other classes c_1 and c_2 by the site owner. This effectively removes the class c, but since when a component is moved out of a class its migration history is retained, during retrieval c_1 and c_2 can still be found with relevance about 0.707α times the relevance of c. Thus, the results of a query against a site will vary only moderately even if the classification structure of that site has been altered wildly.

6 Browsing

Internally the faceted classification structure is represented as a relational database table, with the facets as the columns and the component classes as the rows. The browse tool works by representing this faceted structure as a tree. A tree node in this representation stands for a subtable that is constructed

by performing appropriate selection and projection operations against its parent table. The root node stands for the whole table. For example, suppose there are four facets, with the following table for the faceted structure:

task	target	pricing	source
edit	database file	free	yes
edit	text file	shareware	yes
browse	database file	free	yes
browse	web	free	no

an expansion on the facet "task" yields two subnodes "edit" and "browse" of the root, as illustrated in Figure 3 (a). A second expansion against the "edit" node yields another two nodes "database file" and "text file," as in Figure 3 (b). By default, the facets are expanded in their original order, but the user can also specify the expanding facet for any node. For example, the user can choose to expand the root node with "source" instead of "task," and then the "yes" node can be expanded further to explore all and only those components delivered with source code, as shown in Figure 3 (c). The faceted structure can thus be expanded in various ways, making it more flexible than a fixed hierarchy. The expanding facets can be optionally displayed to the right of the node name (term) in parentheses. Nodes at the same level can be assigned different expanding facets, as shown in Figure 3 (d).

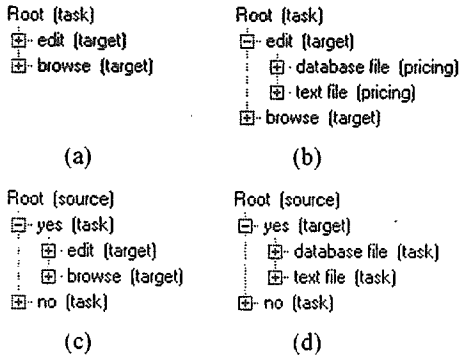

Figure 3: Browsing hierarchies built over the faceted classification structure.

Nodes in an expansion tree correspond to classes in the faceted classification structure. For example, in Figure 3 (a) the node "edit" corresponds to the class with the descriptor (task = edit), or more formally, the descriptor (task = edit, target = *, pricing = *, ...). Note that a component with descriptor (task = edit, target = text file) does *not* belong to this class, although it as a query can match the descriptor (task = edit). That descriptor, however, belongs to the class corresponding to the node "text file" in Figure 3 (b). The root is a special node, which corresponds to an unindexed class, that is, a class associated to no descriptor. It can be used as a temporary storage for new components.

The browse tool is used for both finding components and managing a site, provided the user has appropriate privilege. Most of the modification operations to the faceted classification structure can be performed directly with mouse actions in the browse tool window. For example, a component can be moved or copied from one class to another by drag-and-dropping the component to the node corresponding to the destination class. This effectively changes or adds a descriptor associated to the component. Certain operations involving the facets, on the other hand, must be performed with menu commands. Examples are addition, deletion and reordering of facets.

Move and copy operations can also be performed against nodes, but care must be taken in this case. Some node operations are forbidden in order to reserve the faceted structure. For example, moving the "show" node to under the "edit" node in Figure 4 (a) is invalid because this will cause a conflict in expanding facets, as illustrated in Figure 4 (b). In order to examine the validity of a node operation, we first define the path facets $F(q)$ of a node q in the tree as the set of expanding facets of the nodes from the parent of q up to the root node. For example, the path facets of the "text file" node in Figure 3 (a) is {task, target}. Then a child of node q can be successfully copied or moved to become a child of node q' if (1) q and q' have the same expanding facet, and (2) $F(q') \subset F(q)$.

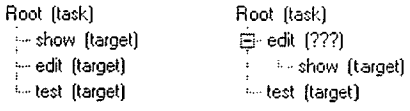

Figure 4: An invalid node operation. Moving the "show" node to under the "edit" node is invalid.

7 Implementation

7.1 The Query Tool

The appearance of the query tool is shown in Figure 5. Several queries can be issued at the same time by the user, as long as they are separated by the "|" character. In Figure 5 two queries, namely (function = edit, target = text) and (function = delete, target = file), are issued. The results of those two queries are merged together, again ordered by relevance, and listed in the "Results" area. To control the number of components retrieved, two parameters *maximum components* (denoted by N) and *minimum relevance* (denoted by L) can be set, and only the first N components with relevance values no lower than L are listed. The user also has to specify the target sites. For simplicity only one site can be queried against at a time in the current implementation. Whenever a component is found during retrieval, its related classes can be checked automatically to find more components. The user can specify to search for these related classes in all other sites, in part of them, or in the local site only. If the user choose to trace only a number of sites, these sites can be specified below the target site. In Figure 5 the target site is "localhost" (the local site) and the additional sites are "gaia.ee.ntu.edu.tw" and "ibmsrv.cc.nthu.edu.tw."

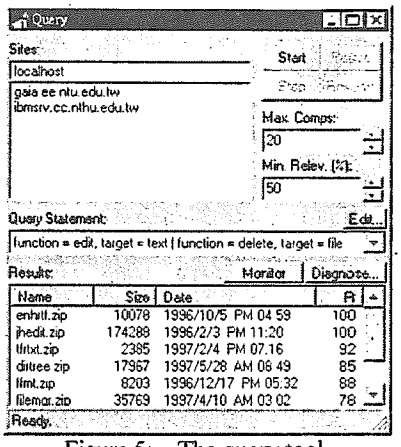

Figure 5: The query tool.

7.2 The Browse Tool

The appearance of the browse tool is shown in Figure 6. The window is divided into three panes. To the left is the *tree pane*, in which the classification structure is shown as a tree.

The upper-right one is the *node pane*; below it is the *component pane*. They show the sub-nodes and components, respectively, belonging to the selected node in the tree pane. The user can expand, collapse or select a tree node in the tree pane, and the contents of the other two panes will be updated accordingly.

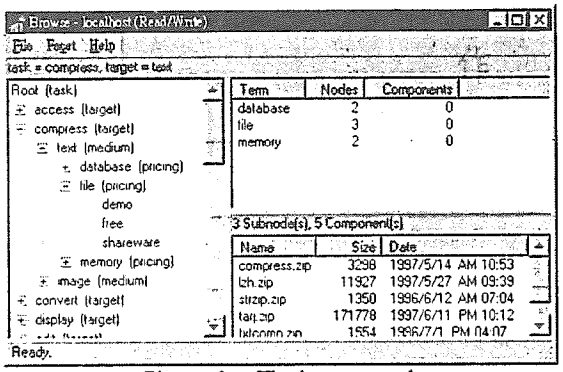

Figure 6: The browse tool.

7.3 General Usage

These two tools are used side-by-side to facilitate searching in component repositories. If the user has a clear idea about the functionality of the desired component, the query tool can be used. If a component matching the requirement is found on the remote site, and if the user has the manager privilege on the local site, the found component can be added to the local site by opening the local site in read/write mode with the browse tool, and then drag-and-dropping the component from the query window to a node in the browse window. The retrieved component is then indexed with descriptor corresponding to the destination node.

In case the user is not familiar with the vocabulary of the accessed site, or when the requirement is not clear, the browse tool can be of greater help in searching components than the query tool. The site to be searched can be opened with the browse tool in read-only mode. When a component is found, the local site can be opened read/write in another browse tool window, and the found component with can be stored the new retrieved.

8 Conclusion and Future Directions

In this paper, our approach towards a global development environment for component-oriented software reuse is presented. In this approach software assets are fully distributed into small repositories that can be easily managed by end-users. The classic multi-facet classification method is adopted and enhanced with techniques such as degradation functions. Browsing hierarchy can be dynamically built and manipulated over the faceted structure so that the user can navigate and manage a repository intuitively and lucidly. Components of similar functionality can be found efficiently with the migration history technique, even if they reside in different repositories or if the classification structure has been wildly changed in the source sites. Finally, a set of tools with user-friendly graphical interface have been built to facilitate access to repositories.

In the current implementation only one view is allowed in a site. This may cause inconvenience and inefficiency if the user is willing to include a wider range of components in a site. A hybrid architecture, which allows multiple views to coexist in a site by combining enumerated [16] and faceted approaches, has been examined and will be implemented into future versions of the Uranus system. Another possible improvement is to replace GUIDs with content-derived names (CDNs) described in [17]. Perhaps the most important step in our future research is to put the system into real-world use. Conducting such a field test is quite difficult, but user feedback will be of great importance to future development.

Our system is intended to work as a fundamental utility for creating an open software market in which circulation and reuse of software parts are strongly encouraged. We do not expect our system, or any system of this kind, to solve all software reuse problems, but we believe that if used properly the distributed repositories approach can greatly improve the overall productivity of the software industry.

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, 1993.

[2] C. W. Krueger, "Software Reuse," *ACM Computing Surveys*, vol. 24, no. 2, pp. 131–183. 1992.

[3] S. Henninger, "Information Access Tools for Software Reuse," *Journal of Systems Software*, vol. 30, pp. 231–247, 1995.

[4] W. Tracz, "Software Reuse Maxims," *ACM Software Engineering Notes*, vol. 14, no. 4, pp. 28–31, 1988.

[5] G. Arango, "Software Reusability and the Internet," *ACM Symposium on Software Reusability*, pp. 22–23, Seattle, WA, USA, April 1995.

[6] S. Browne, J. Dongarra, S. Green and K. Moore, "Location-Independent Naming for Virtual Distributed Software Repositories," *ACM Symposium on Software Reusability*, pp. 179–185, Seattle, WA, USA, April 1995.

[7] S. V. Browne and J. W. Moore, "Reuse Library Interoperability and the World Wide Web," *ACM Symposium on Software Reusability*, pp. 182–189, MA, USA, April 1997.

[8] J. S. Poulin and K. J. Werkman, "Melding Structured Abstracts and the World Wide Web for Retrieval of Reusable Components," *ACM Symposium on Software Reusability*, pp. 160–168, Seattle, WA, USA, April 1995.

[9] M. Davis, "On Practicality of Domain-Specific Languages and Analysis and Multifaceted Reuse Libraries," *ACM Symposium on Software Reusability*, pp. 104–109, MA, USA, April 1997.

[10] H. Mili, E. Ah-Ki, R. Godin and H. Mcheick, "Another Nail to the Coffin of Faceted Controlled-Vocabulary Component Classification and Retrieval," *ACM Symposium on Software Reusability*, pp. 89–98, MA, USA, April 1997.

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

[12] L.-Y. Chou and C.-C. Yang, "Automatic Construction and Maintenance of Software Hierarchies for Improving Reusability," *Proc. National Computer Symposium*, ROC, 1993.

[13] H. Mili *et al.*, "Practitioner and SoftClass: A Comparative Study of Two Software Reuse Research Projects," *Journal of Systems Software*, vol. 25, pp. 147–170, 1994.

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

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

[16] 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.

[17] 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.