Ranking Significance of Software Components Based on Use Relations

Katsuro Inoue, *Member*, *IEEE*, Reishi Yokomori, *Member*, *IEEE*, Tetsuo Yamamoto, *Member*, *IEEE*, Makoto Matsushita, and Shinji Kusumoto, *Member*, *IEEE*

Abstract—Collections of already developed programs are important resources for efficient development of reliable software systems. In this paper, we propose a novel graph-representation model of a software component library (repository), called *component rank model*. This is based on analyzing actual usage relations of the components and propagating the significance through the usage relations. Using the component rank model, we have developed a Java class retrieval system named SPARS-J and applied SPARS-J to various collections of Java files. The result shows that SPARS-J gives a higher rank to components that are used more frequently. As a result, software engineers looking for a component have a better chance of finding it quickly. SPARS-J has been used by two companies, and has produced promising results.

Index Terms—Component rank, graph representation model, reuse models, program analysis, reusable libraries.

1 Introduction

Computer systems are becoming core infrastructures of effective and efficient activities of everyday life. The software that exists in these computer systems is becoming ever larger and more complex, and demand for high software quality is becoming stronger. One promising approach to the efficient development of quality software is to leverage software reuse.

A lot of research on structuring and retrieving software libraries (repositories) for reuse has been performed [10], [14], [16], [19], [21], [23]. However, we do not know much about successful cases, in the sense that library reuse is widely prevalent in software development organizations. Library reuse is vital for efficient development of quality software in the organization.

Mili et al. have extensively investigated and precisely classified a wide variety of research on retrieval of software libraries, and have shown the nature and various characteristics of classified technologies [17]. Their results suggest that a promising approach to a practical reuse system is to employ the information retrieval method based on a textual analysis of software. This method can be highly automated with a low operational cost, and we can easily apply various techniques developed for natural language and HTML documents. However, since the repository and retrieval structures are generally very simple, we usually get a broad result for a query. Thus, it is essential to introduce a mechanism to narrow the query result.

In this paper, we propose a novel ranking method to narrow retrieved software components from reusable libraries. We define a *component rank model* based on a graph representation scheme of the component library [9]. In this model, a collection of software components is represented as a weighted directed graph, i.e., the nodes of the graph correspond to components and the edges linking the nodes correspond to cross component usage. Similar components are clustered into one node so that the effect of duplicated components is removed. The nodes in the graph are ranked by their weights, which are defined as the elements of the eigenvector of an adjacent matrix for the directed graph. The resulting rank, named *component rank*, is used to prioritize the query result so that highly ranked components are quickly seen by the user. The idea behind component rank originates from computing impact factors (called *influence weights*) of published papers [20]. This approach has been extended to ranking Web documents on the Internet [18].

Using the component rank, we have developed a component search system, named SPARS-J, which treats the source files of Java classes as components. This system has been applied to various collections of Java programs, such as JDK, programs downloaded from the Internet, and business applications from two companies.

The results show that a class frequently invoked by other classes (such as those that implement fundamental and standard data structures) generally has a high rank, and that nonstandard and special classes typically have a low ranking. Two companies use SPARS-J for automatic management of their software assets, and SPARS-J shows very promising results.

In Section 2, we propose a component rank model. Section 3 shows the Java component search system SPARS-J based on the component rank model. The results of

- *K. Inoue, R. Yokomori, M. Matsushita, and S. Kusumoto are with the Software Engineering Laboratory, Department of Computer Science, Graduate School of Information Science and Technology, Osaka University, 1-3, Machikaneyama-cho, Toyonaka-city, Osaka, 560-8531, Japan. E-mail: {inoue, yokomori, matusita, kusumoto}@ist.osaka-u.ac.jp.*
- *T. Yamamoto is with the Department of Computer Science, College of Information Science and Engineering, Ritsumeikan University, Noji Higashi 1-1-1, Kusatsu City, Shiga 525-8577, Japan. E-mail: tetsuo@cs.ritsumei.ac.jp.*

Manuscript received 19 May 2004; revised 15 Dec. 2004; accepted 20 Dec. 2004; published online 20 Apr. 2005.
Recommended for acceptance by W. Frakes.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0095-0504.

0098-5589/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society

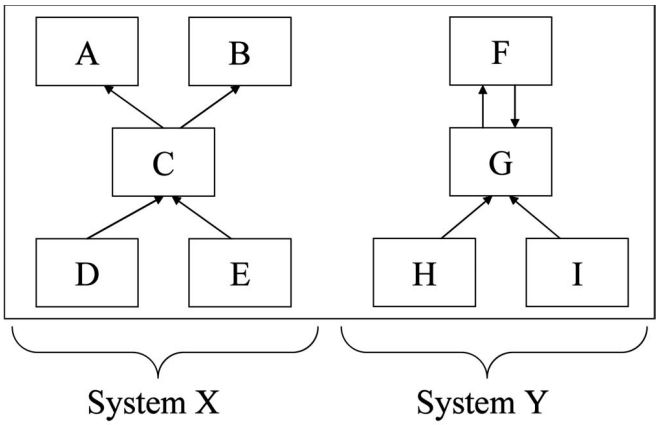

Fig. 1. An example of component graph.

applications are presented in Section 4. Various issues related to the model and the implementation are discussed in Section 5, and related works are presented in Section 6. Finally, we conclude our discussion with future work in Section 7.

2 Component Rank Model

2.1 Component Graph

Software systems are modeled by a weighted directed graph, called a *Component Graph*. A node in a graph represents a software component, and a directed edge e_{xy} from node x to y represents a *use relation*, meaning that component x uses component y.

However, we do not restrict our discussion to a specific kind of component in the graph. A component may be a source-code module, a linked library, or one section of a document. The following discussion will hold for any kind of node.[1] Also, the use relation is left unspecified here for the extendibility of the model. For example, we can consider a method call, a class inheritance, a reference in a library, or a dependency between documents as a use relation. In Section 3, we will explain our concrete implementation for those abstracted nodes and edges, such that the components are Java class files and that the use relations are the class inheritance, method invocation, abstract class implementation, and so on.

A system can be modeled by a component graph that is weakly connected (assuming that there is no redundant component). A collection of software systems, that is, a software library, is also modeled with one component graph. This graph is disconnected if there are no shared components.

Fig. 1 shows a component graph for two software systems X and Y. X consists of five components A - E, and Y consists of four components F - I. This graph also illustrates that component C uses both A and B, and D and E use C. Also, H and I use G, and G and F mutually use one another.

1. Currently, we assume that nodes in one graph are of a single kind only, although we could extend it to allow more than one kind of node.

2.2 Weight of Node

Each node v in component graph G has a nonnegative weight value $w(v)$ where $0 \leq w(v) \leq 1$.

Definition 1 (Total Weights of Nodes). *For simplicity in the following calculation, we assume that the sum of the weights of all nodes in G is 1, i.e.,*

$$\sum_{v \in G} w(v) = 1.$$

Computing the weight for each node under the computation policies described below is our objective of this work. The order of the nodes sorted by the weights is called the *component rank* of the components.

We introduce several definitions to define $w(v)$ for component graph $G = (V, E)$.

Definition 2 (Weight of Edge). *For computation of the weights of nodes, we introduce the weight $w'(e_{ij})$ of an edge $e_{ij} = (v_i, v_j)$, such that*

$$w'(e_{ij}) = d_{ij} \times w(v_i).$$

Fig. 2a depicts this definition. Here, d_{ij} is called a *distribution ratio*, where $0 \leq d_{ij} \leq 1$ and the total of d_{ij} for each j is 1. If there is no edge from v_i to v_j, $d_{ij} = 0$. In Section 2.4, we will discuss the case that v_i has no outgoing edge at all. The distribution ratio d_{ij} is used to determine the forwarding weights of v_i to an adjacent node v_j.

Definition 3 (Weight of Node). *The weight of a node v_i is defined as the sum of the weights of all incoming edges e_{ki}, such that*

$$w(v_i) = \sum_{e_{ki} \in \mathrm{IN}(v_i)} w'(e_{ki}).$$

Here, $\mathrm{IN}(v_i)$ is the set of the incoming edges of v_i. Fig. 2b shows this definition.

2.3 Computation of Weights

Based on these definitions, we have $n(= |V|)$ simultaneous equations for $w(v_i)$,

$$w(v_i) = \sum_{e_{ki} \in \mathrm{IN}(v_i)} d_{ki} \times w(v_k).$$

Assume that W is a vector of the node's weights,

$$W = \begin{pmatrix} w(v_1) \\ w(v_2) \\ \vdots \\ w(v_n) \end{pmatrix}.$$

Also, D is a matrix of the distribution ratios,

$$D = \begin{pmatrix} d_{11} & d_{12} & \dots & d_{1n} \\ d_{21} & d_{22} & \dots & d_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n1} & d_{n2} & \dots & d_{nn} \end{pmatrix}.$$

Therefore, the simultaneous equations can be rewritten by

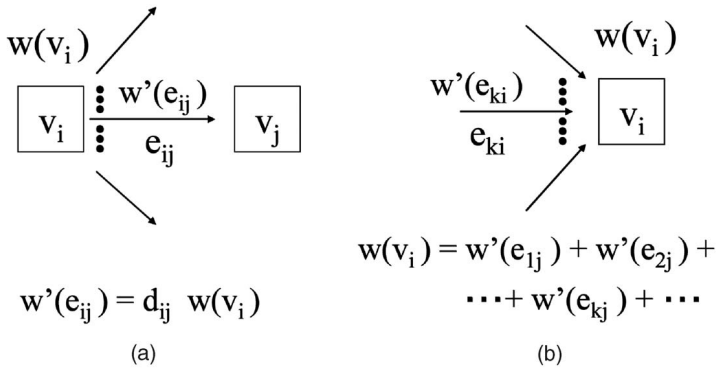

Fig. 2. Definition of weights. (a) Weight of edge. (b) Weight of node.

$$W = D^t W, \tag{1}$$

where D^t is the transposed matrix of D.

Together with Definition 1, (1) can be solved by computing the eigenvector with eigenvalue 1. Instead of computing the eigenvector, we can compute the weights of each node by a repeated computation (named a *power method*) such that we give initial ad-hoc weights to each node (e.g., $1/n$ to each node), and then propagate them to adjacent nodes through directed edges. The weights are repeatedly recomputed until the all weights become stable.

Fig. 3 shows a component graph with computed weights, where v_1 has two outgoing edges, and weight 0.4 is evenly divided between two outgoing edges with 0.2 each (i.e., $d_{12} = d_{13} = 0.5$). Here, v_3 has two incoming edges, each with a weight of 0.2, so that the weight of v_3 is 0.4.

If we assume that the movement of a software developer's focus on the target components is represented by a probabilistic state transition, the component graph is understood as a Markov chain model [25]. Thus, computing the weights of the nodes in the graph corresponds to attaining a stationary distribution of the chain.

2.4 Convergence of Computation

In the case that a node v_i has no outgoing edge, all distribution ratios d_{ix} become zero. This does not satisfy the requirement of the total sum of the distribution ratio. Also, if the graph is not strongly connected, as shown in Fig. 4a, the weight of v_1 is repeatedly added to the weight of v_2, causing nontermination in the repeated computation.

In these cases, the eigenvector cannot be solved either. To guarantee the solution of the simultaneous equations, i.e., the termination of the power method, we introduce pseudo use relations between all nodes, as shown in Fig. 4b, and define an amended distribution ratio for each node as follows:

Definition 4 (Amended Distribution Ratio).

$$d'_{ij} = \begin{cases} p \times d_{ij} + (1-p)/n & \text{if } v_i \text{ has outgoing edge } e_{ij}, \\ 1/n & \text{if } v_i \text{ has no outgoing edges.} \end{cases}$$

Here, $p(0 < p < 1)$ is the ratio between the weight of real use relations and that of pseudouse relations, and we generally employ a fairly large value, such as 0.85 as discussed in Section 5.1.3. By using d'_{ij} instead of d_{ij}, all elements in the matrix become positive. Since such a matrix is an irreducible one, the dominant eigenvector is uniquely defined. This pseudouse relation can be considered as a possible implicit reference of a component to all components (including itself).

2.5 Clustering Components

Software systems are often created by reusing already developed components. Some components are simply copied from previous systems, while some are constructed from those with minor or major modifications. In the above-mentioned method, those reused components are represented as multiple nodes in a component graph, and their weights are computed independently. We want to identify the similarity of components and feed back the similarity information to the computation in order to obtain more practical weight values.

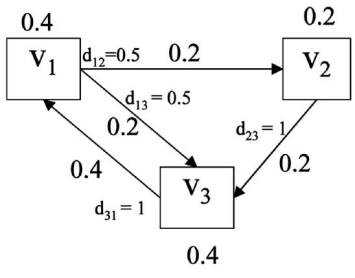

Fig. 3. An example of stable weights assigned to nodes and edges.

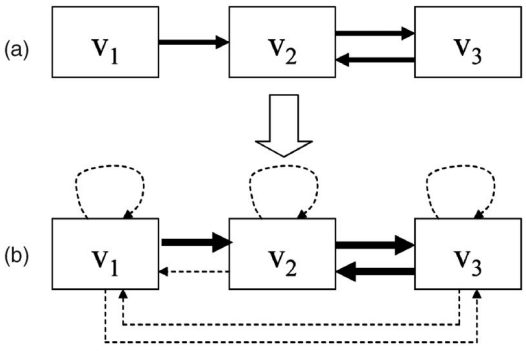

Fig. 4. Introduction of pseudo use relation for convergence.

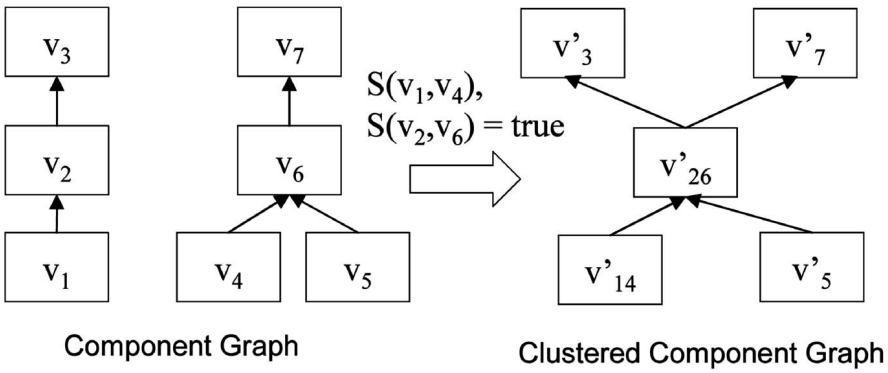

Fig. 5. An example of a clustered component graph.

Assume that we could determine similarity of any two nodes (components) v_1 and v_2 by a logical function $S(v_1, v_2)$. If v_1 and v_2 are similar components, $S(v_1, v_2)$ is true, otherwise it is false. The relation defined by S is an equivalence relation on V. This equivalence relation partitions V into a set of equivalence classes, composing the quotient set V'.

We define the *clustered component graph* $G' = (V', E')$ for G as follows:

Definition 5 (Clustered Component Graph). *$G' = (V', E')$ such that V' is the quotient set of V, and $E' = \{(v'_i, v'_j) | (v_i, v_j) \in E\}$, where v'_i and v'_j are equivalence classes involving v_i and v_j, respectively.*

Fig. 5 shows an example of a clustered component graph. Here, we assume that v_1 and v_4 are similar, and also v_2 and v_6 are similar. Then, two weakly connected subgraphs in the component graph are merged into one weakly connected graph as the clustered component graph. As this example shows, there are cases such that several independent subgraphs in the component graph are merged into one, and the characteristics of each subgraph can be propagated to other subgraphs.

Using the clustered component graph instead of the component graph, we will compute the weights of each node with the above-mentioned approach, including amendment of the distribution ratio. For simplicity, hereafter we use *component graph* for *clustered component graph* if there is no ambiguity.

3 SPARS-J

3.1 Implementation of Component Rank

Based on the component rank model discussed in Section 2, we have designed and implemented SPARS-J (Software Product Archiving and Retrieving System for Java) to compute the component rank and to search components for Java programs. The following are features of the implementation:

- **Component**: A Java class or interface source code is considered a component. In Java, it is inherently easy to extract components, since each class or interface definition is independently defined as a single file with the `.java` extension. Internal classes are also considered components here.
- **Use relation**: Use relations in the component rank model are class inheritance, interface implementation, abstract class implementation, variable declaration, instance creation, field access, and method invocation. Here, we employ only statically detectable relations from the source programs.
- **Ratio p between real and pseudo use relations**: We use p=0.85, which means that the total weight value 0.15 is assigned to the pseudo use relations. We will discuss this value in Section 5.1.3.
- **Distribution ratio**: The distribution ratios of the real use relations, which emanate from one node, are equal values. For example, if a node has five outgoing edges, the distribution ratio evenly divided for each edge is 0.17. The total of five edges is 0.85, and the remaining 0.15 is the total of the pseudouse relations.
- **Similarity S**: A similarity measurement tool, proposed in [13], is used as the similarity function S. The tool measures metrics of a pair of components from two viewpoints. One is a complexity metric such as the number of methods and the cyclomatic number. Another is a word distribution metric that indicates the frequency of each token in a component. We compare metric values between two components and, if the differences are less than the threshold values heuristically obtained by our experience, we consider that these two components are similar.[2]

3.2 Architecture of SPARS-J

Fig. 6 shows the architecture of SPARS-J. The following outlines the process steps of SPARS-J:

1. Components, i.e., `.java` files, are collected from various kinds of Java distribution packages.
2. The syntax analyzer parses the components and extracts the use relations mentioned above.
3. The similarity of two components is computed. For efficiency, we employ a hash technique to search similar components.

2. Examples of the threshold values are that the difference of the word distribution metric values is less than 3 percent, the difference of the number of method call statements is less than 2, and so on. Intuitively speaking, if two components share more than 90 percent of source code lines, they are determined to be similar ones.

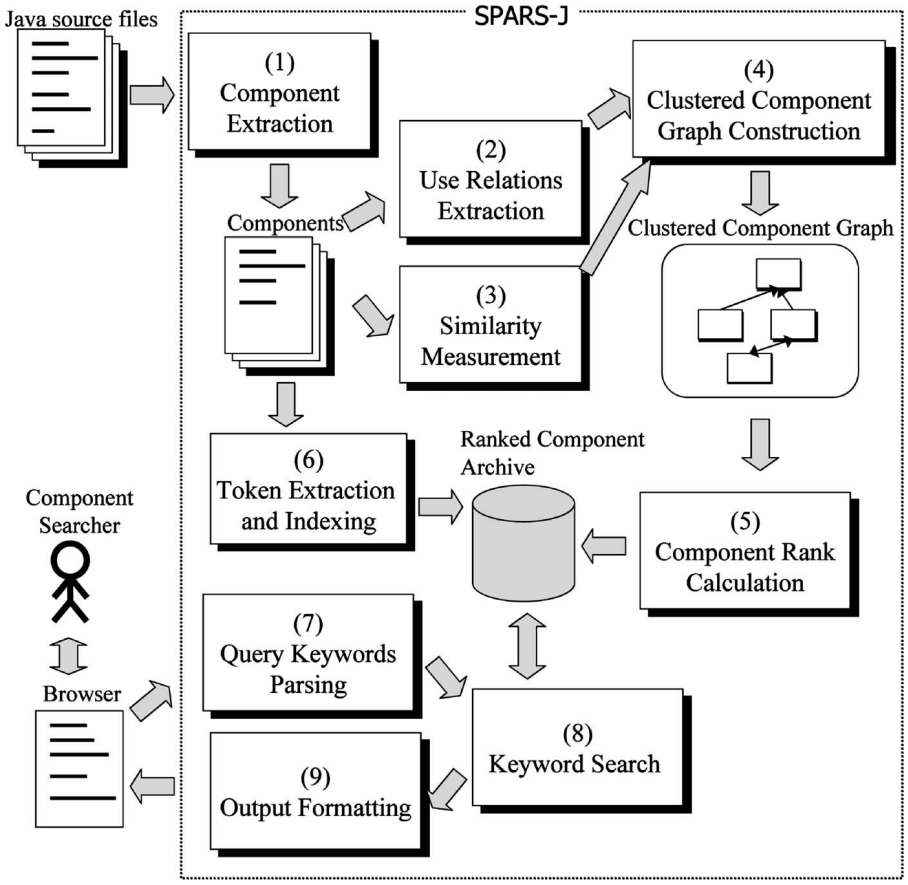

Fig. 6. Architecture of SPARS-J.

4. Using the results of (2) and (3), a clustered component graph is constructed.
5. Node weights are computed by the power method. This method is mostly faster than solving the eigenvector with a math library. The resulting weights are sorted and the component rank is obtained. The rank is stored into the ranked component archive along with the original source code.
6. Various types of tokens in the components are extracted, and the components are indexed by these tokens.

The process Steps 1-6 (named *preprocess steps*) are performed noninteractively before we start the search. The following Steps 7-9 are executed interactively with the component searcher.

7. The query keywords are parsed and forwarded to the keyword at search step.
8. The components containing the same token as the query keywords are searched for archives. All tokens, both in the source code and comments, are explored.
9. The resulting components are listed by the order of the component rank and printed out as an HTML document.

Fig. 7a shows an example screenshot of the resulting component list for given query keywords. The details of a component can be seen by clicking an item on the list, as shown in Fig. 7b. On this screen, we can obtain various views of the component, such as its source code (A), similar components (B), components that use this component (C), components used by this component (D), metrics values of the component (E), and others.

4 Experiments

4.1 Application to JDK 1.4.2

All source programs in the Java 2 Software Development Kit, Standard Edition 1.4.2[3] (simply referred to JDK 1.4.2 here), are the target of the first application. The kit comprises 4,257 files totalling 1,290,000 LOC in Java. These files are the definitions of about 6,100 classes, and those classes are essential for various Java applications.

The highest ranked class is `java.lang.String`, which is the general class for string. Since this class is used throughout the program, we consider it natural for that class to be ranked at the top.

Other highly ranked classes are also fundamental ones that are possibly invoked or inherited from many other classes. The second-ranked class, `java.lang.Object`, is the super-class of any class in Java, meaning that this class is used directly or indirectly by any other class. There are

3. Java 2 Software Development Kit, Standard Edition 1.4.2, http://java.sun.com/j2se.

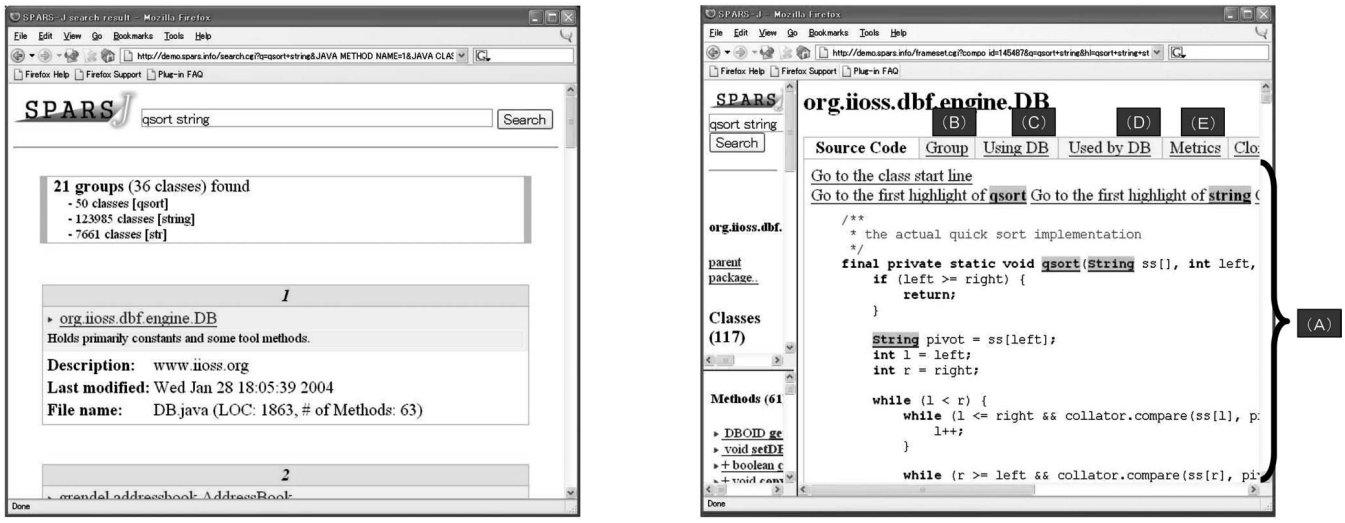

Fig. 7. Screenshots of SPARS-J. (a) Listing result of a query with "qsort string." (b) A view of a component.

1,147 classes with the lowest (4,966th) rank; these classes are not used at all by any other JDK classes.

The overall result of component ranking for JDK 1.4.2 matches our intuition such that very general and core classes are ranked high (significant), and specific and independent classes are ranked low (insignificant).

4.2 Application to Large Collection of Publicly Available Components

We have collected publicly available Java source codes from various projects, i.e., JDK 1.4.2, open-source software found at SourceForge.net,[4] and other Internet sites. In total, we have collected 140,000 Java files. SPARS-J analyzed these files and created a huge database with 180,000 Java classes and their component ranks.

The top-ranked class is `java.lang.String`, and the second is `java.lang.Object`. In comparison with the component rank for the collection of only JDK, the higher-ranked components are almost the same. The highest-ranked non-JDK component is `org.w3c.dom.Node` (86th), which expresses a node in the DOM tree structure of XML documents. Since higher-ranked components are of JDK, we could recognize that JDK is closely related to any type of Java software.

This result might raise the concern that we would always obtain JDK components as the result of a component search. However, by giving proper keywords, we can obtain useful results as the following section shows. Moreover, SPARS-J contains a feature to specify the search space, by which we can remove JDK or other packages if needed.

4.3 Searching Java Classes

To investigate the capability of SPARS-J as a search engine for Java classes, we wanted to compare it with other Java class-search systems having similar features to SPARS-J. However, we were unable to obtain publicly available and practically applicable systems. Therefore, as an alternative approach, we have compared SPARS-J with two general search engines, Google[5] and Namazu.[6]

Google is a very powerful Web search engine used for a whole variety of purposes; actually, in Google, there are many Web pages that contain Java source code. Recently, many developers have been using Google to get various kinds of information about software. We will witness the usefulness of SPARS-J compared with Google in the purpose of finding Java code for a simple reference. On the other hand, Namazu is a reliable and easily used full-text search system using the TF (Term Frequency)-IDF (Inverse Document Frequency) method [22]. This comparison is made to highlight the effectiveness of the SPARS-J mechanism compared with a simple full-text search system.

We used the aforementioned database of 180,000 publicly available Java classes, and also constructed a database for Namazu from the same data set. Since we cannot change the database for Google, we simply used Google as it is by always giving each query keyword associated with the extra keywords "java" and "source." We executed various searches on three systems by employing keywords, obtaining the precision for each of the top 10 results,[7] as shown in Table 1.[8] These keywords were chosen for various search contexts, such as reference to a tool or a software (Q3, Q5, Q7, Q8), reference to an algorithm (Q1, Q2, Q6), and reference to an example of a library (Q4, Q9, Q10). We think that these contexts are generic enough to cover expected use of our system.

We performed a paired-difference t-test among the three systems' precisions, and have confirmed that the differences between SPARS-J and Google, and between SPARS-J

4. SOURCEFORGE.net, http://sourceforge.net

5. Google, http://www.google.com.

6. Namazu, http://www.namazu.org/index.html.en.

7. We generally expect one desirable item listed in the first page of the search result [24]. Also, the recall is not investigated, since we usually do not want all of the correct answers. We are satisfied with several correct answers listed at the beginning of the list.

8. For Google and Namazu, similar classes were clustered by hand. Also, we allowed to look for expected classes through one link from the pages returned by Google.

TABLE 1
Precisions of Each System

Query	Keywords	SPARS-J	Google	Namazu
Q1	quicksort	1	0.7	0.9
Q2	binarysearch	1	0.4	0.6
Q3	clock applet	0.5	0.3	0.4
Q4	applet textarea	0.4	0.3	0.6
Q5	random number generate	0.4	0.1	0.3
Q6	stack push pop	0.2	0	0.1
Q7	chat server client	0.9	0.3	0.4
Q8	classfile dump	1	0.1	0.2
Q9	zip deflate	0.6	0.4	0.4
Q10	write outputstream read inputstream	0.5	0.4	0.7
Ave.		0.65	0.3	0.46

and Namazu, are significant at the 5 percent level. That is, SPARS-J is superior to the other two systems with respect to precision.

4.4 Evaluation of Component Rank

As a measure of the component rank's effectiveness, we investigated the distance between the component rank and the other orders. The components containing query keywords are ordered by three methods: 1) the component rank, 2) Namazu using full-text search with the TF-IDF method, and 3) hand-made ordering by software experts to determine the significance of the components.

The distances of two orders, i.e., case 1 (1-3) and case 2 (2-3), are measured with the *Normalized Distance-based Performance Measure NDPM* method [28]. The NDPM value becomes smaller if two orders come closer to each other. The NDPM values were obtained for the same query keywords in Table 1. We tested the difference of average values of case 1 (0.143) and case 2 (0.178), and confirmed the difference with a 5 percent significance. This means that the component rank provides closer ordering to the software expert ordering compared with the order obtained by the simple text handling method.

4.5 Case Studies in Two Companies

4.5.1 Case Study 1

Daiwa Computer, located in Osaka, Japan, is a software company with about 180 engineers. In that company, various types of software engineering technology are actively studied and introduced. The company holds an ISO9001:2000 certificate and CMM Level-3 assessment. A shared framework for Java applications has been constructed in the company and various business applications have been developed on it.

Using this framework, five Web-based data management applications have been developed. Those applications have similar features but include explicit differences in their implementation, such as databases and their interfaces (SQL, DB2, Oracle, ...). These five applications and the framework itself form the target software library of the ranking. The number of components in the framework is 250, and the overall library contains 1,538 components in total, which are clustered into 339 nodes.

Fig. 8 shows the ratio of the framework classes found in the top of the resulting list. The X axis represents the number of highest-ranked classes. This graph indicates that

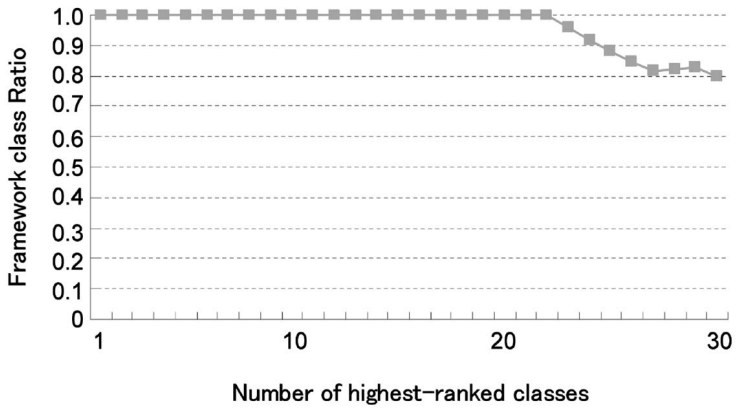

Fig. 8. The framework class ratio in the top result.

TABLE 2
Used Queries in Case 2

♯	Objective	Keyword	Precision of top 10
1	Find classes affected by deleting isAdministrator method	isAdministrator	1
2	Find methods using an exception for an error message	ProcessStatus-AlreadyChanged-Exception	1
3	Find an implementation of 'check' method	check	0.7
4	Find a method for checking a schedule	schedule	0.8
5	Find classes which use table BTA.	BTA 002	0.5
6	Find a typical way of obtaining SGNCompanyCode	SGNCompanyCode	0.75

the classes with the first to 21st ranks are of the framework, and also shows that nonframework classes gradually appear after those highly ranked framework classes.

We investigated the highly-ranked classes and found that those classes are the definition of data structures and their containers. For example, the first-ranked class is the definition of a record class for database management. These results confirm our approach, i.e., it is easy to identify core and fundamental components by their ranking.

4.5.2 Case Study 2

Next, we applied SPARS-J to the information system of Suntory Limited, which is Japan's leading producer and distributor of alcoholic and nonalcoholic beverages, where hundreds of Java applications have been developed for various activities such as sales, deliveries, accounts, and so on. The company is interested in SPARS-J for the maintenance and reuse of its applications. That is, the company wants to identify, by component rank, important components that have been reliable and reusable frequently, and to preserve those identified components as company-wide assets.

To evaluate SPARS-J, the company provided about 2,400 components (classes) that are used in the many application programs developed in the company. Using this collection, seven software engineers and software managers evaluated SPARS-J for two weeks [26]. They are in charge of standardization and quality management in the company, and are knowledgeable about these 2,400 classes. They gave various queries to SPARS-J and checked the results. Table 2 shows a part of the evaluation, i.e., objective of each query, query keyword, and the precision of the top 10 highest-ranked classes. The correctness of these ten classes was determined by those engineers and managers. As shown in the table, those precision values are fairly high, demonstrating the capability of SPARS-J for finding the required classes.

At the end of the evaluation, a questionnaire was collected as a qualitative evaluation of SPARS-J. The questionnaire asked the evaluators about the usefulness and effectiveness of individual and overall features of SPARS-J. Table 3 lists the questionnaire items and the summarized result. Each item is ranked by a five-level rating from A to E, and the number of each rate is counted.

This result shows that the SPARS-J was supported by the engineers and managers. The display features provided by SPARS-J (such as using and used-by relations) score highly, as the ranking feature does. Furthermore, some engineers reported that it is easy to grasp the structure of the application and to perform an impact analysis for modification of a component.

Based on these results, we can state that SPARS-J is applicable and useful for actual software maintenance and reuse activities. The company is now using SPARS-J in their daily activities as its main software repository.

5 Discussion

5.1 Component Rank Model

5.1.1 Weight Computation

In our component rank model, the weights of nodes are propagated to other nodes through edges. A simpler model we can easily consider would have the node weight determined by the number of incoming edges. This alternate model does not require computation of the convergence for eigenvector.

However, this alternate model is very fragile and affected by local special references. Consider that a component A is used once by each component B1 through B10, and B1 to B10 are not used by any component at all. In the alternate model, the weight of A becomes 10. In the component rank model, the weight of A is relatively determined by other components, but A is not ranked as highly as the alternate model, since the weights of B1 to B10 are relatively low. Thus, we consider that our model is very stable for such local references.

This argument is similar to a comparison of Web search engines between Google and other simple search engines that counts only incoming references [18]. The simple engines could be easily affected by intentionally made local reference links, but Google is hardly influenced by such links.

5.1.2 Clustering Policy

The clustering of similar nodes is an important characteristic of our component rank model. If we would model multiple software systems in a component graph without clustering, the graph might be composed of multiple disconnected subgraphs and the weights would be independently computed inside each subgraph, without circulating weights over subgraphs (except for minor interaction by the pseudo use relations). The clustering has an effect that the weights of one software system are propagated to

TABLE 3
Questionnaire Items and Rates

Questionnaire item	♯ of rated Answers					
	A	B	C	D	E	N/A
For evaluating individual features:						
Ranking search result	**3**	2	2	0	0	0
Browsing package trees	**3**	2	2	0	0	0
Grouping similar classes	**2**	**2**	1	1	0	1
Displaying 'using' relation	**6**	0	0	0	0	1
Displaying 'used-by' relation	**5**	0	0	0	1	1
Displaying metrics	1	**2**	0	1	**2**	1
Downloading source code	**3**	0	1	1	1	1
Highlighting found query keywords	**6**	0	1	0	0	0
For evaluation of overall features:						
Do you think that SPARS-J helps to reduce maintenance costs?	**2**	1	**2**	0	1	1
Do you think that SPARS-J helps to improve product quality?	1	1	**3**	0	1	1
Do you think that SPARS-J helps understanding of software assets in the company?	**2**	0	**2**	1	**2**	0
Overall evaluation of SPARS-J	2	**3**	0	1	0	1
Do you want to continue to use SPARS-J?	**3**	2	1	0	0	1

A bold number indicates the mode of the answered rate.

Rate	
A	Best / Yes, very much.
B	Good / Yes, partically.
C	Fair / Don't know.
D	Poor / Don't think so
E	Worst / No, not at all.
N/A	Not answered.

other systems through clustered nodes, resulting in the component rank as a global ranking in all the software systems.

We have taken a clustering policy such that the first similar nodes are clustered, then the weight of each node is computed. An alternate policy would be that we first compute the weight of each node in the original component graph and then cluster the similar nodes by adding their weights.

In our policy, components that are simply copied and used are not counted repeatedly. Consider the simple case shown in Fig. 9 (for simplicity, we do not consider the pseudo use relations here). In this case, components A and B are duplicated. In our policy, the resulting weights for A' and B' are each 1/4, while they are each 1/3 in the alternate policy. This means that, in our policy, the existence of components simply copied does not affect the resulting weight. In the alternate policy, if the number of copied components grows, the resulting weights for those components increase.

We believe that it is important not to count simply duplicated components, but to count modified and labored components.

Consider another case shown in Fig. 10. In this case, component A is reused, but B is replaced with C. In the result, the weight of A' is 2/5 in our policy, which is higher than the 1/3 in the alternate policy. The weight of A' in the alternate policy is 1/3, the same value as the case in Fig. 9, even with its structural change.

These examples clearly demonstrate that duplication with modification is properly counted in our policy, but is not in the alternate policy.

5.1.3 Similarity Function S and Pseudo-Use Relation Ratio p

We have investigated different similarity functions and their threshold values and found that the resulting ranks are fairly insensitive to different threshold values and different comparison algorithms [7], [13].

There are some component pairs that are not merged by specific threshold values but that are merged by slightly looser threshold values; exception handlers in JDK are examples of those. Beacuse we think that those components should not be merged into a single node, we use that threshold.

The ratio p between real and pseudo use relations has been explored in detail [6]. We knew that the resulting weight values are affected by p, but that the resulting component ranks are insensitive to p. The ranks are fairly stable even if we have changed p from 0.75 to 0.95, thus we have chosen $p = 0.85$ here.

5.2 SPARS-J as a Software Component Search Engine

As presented in Section 4, we have performed various experiments for SPARS-J. For these experiments, we used a server system with dual Xeon 2.8GHz processors and 8 GByte of memory. For about 6,100 components in JDK 1.4.2, it took 20 minutes to construct the ranked component archive with all preprocess Steps 1-6 as shown in Fig. 6. Also, it took about two full days for the 180,000 components presented in Section 4.2, resulting in an archive that occupies 5.5 GByte of disk space.

The most time-consuming process steps were those employing relation extraction (Step 2) and token extraction

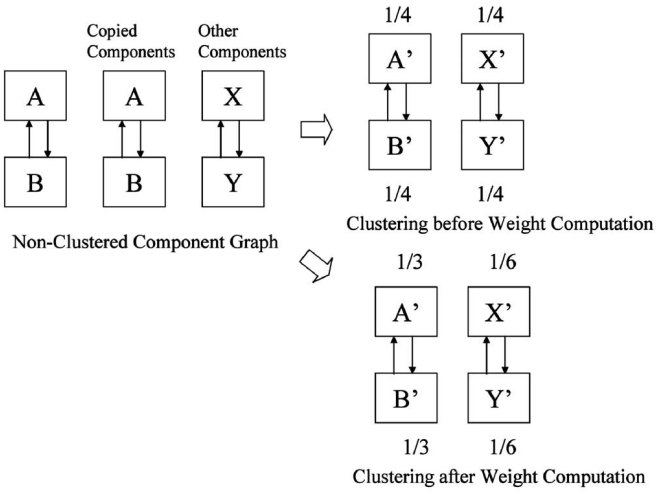

Fig. 9. Effect of simple copied components.

and indexing (Step 6). The component rank calculation (Step 5) required only 15 minutes, even for 180,000 classes.

These preprocess steps are performed only once to construct the ranked component archive. Once the archive is ready, the requested queries are processed almost instantly. For example, the average response time, including all processing time of process steps (Step 7), (Step 8), and (Step 9) for the queries shown in Table 1, is 0.5 sec. Thus, the component searcher can interactively work well with SPARS-J.

The computation time for the preprocess steps grows nonlinearly depending on various factors such as the input file size, the number of components, the number of indexed terms in components, and so on. However, construction of the ranked component archive is a one-time operation and, even for 180,000 components, it finishes in two days. Also, SPARS-J has a feature for adding components to the ranked component archive incrementally, without rebuilding the whole archive.

Since we needed to deal with a huge collection of components, we have used a fairly powerful server machine with an enormous memory space. However, we would think that for ordinary companies with several thousand components, a modest PC will suffice.

We consider that our approach of computing the component rank and using it for software component searches is very practical. As discussed in Section 4.5, SPARS-J with the component rank model is actually used in two companies under practical settings. We are trying to expand such industrial application to other organizations.

We have introduced various features to SPARS-J to improve its performance and usability, such as adding a counting mechanism of keyword occurrences in a component, which is supplementarily used for ranking components with the component rank. In addition, we have developed a package browser by which the component searchers can effectively see around the hierarchy of Java's package names.

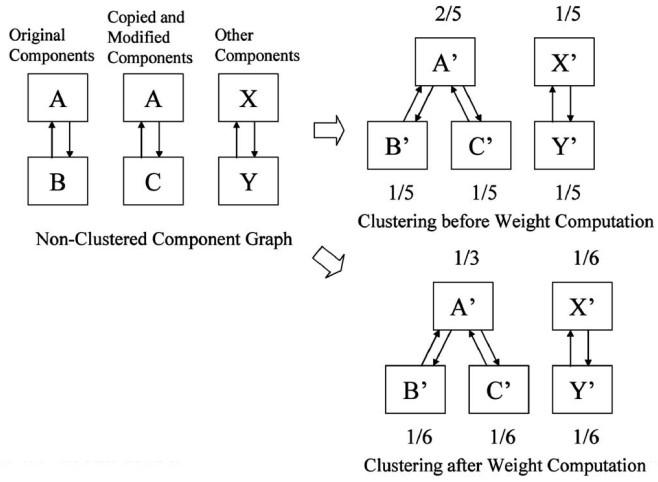

Fig. 10. Effect of copied and modified components.

6 Related Works

6.1 Ranking Model

The component rank model proposed here is based on our original intuition, such that, from a huge number of collections of legacy software, we might be able to effectively search software components, pieces of program codes, program patterns, or abstracted algorithms, by using a similar approach as Google [1], [18] for Web pages, together with various program analysis techniques. Google computes the ranks (called *PageRank*) for HTML documents available on the Internet. The resulting PageRank is practically very useful. A related analyses on the Web links have been discussed in detail in [12].

Google's approach can be considered as an HTML extension of a method proposed for counting the impact of publications, called *influence weight* [20]. The influence weight of a publication is determined by the sum of the weights of incoming references, just as the component rank model does.

A major distinction of our model from PageRank and the influence weight is that our approach explores similarities between components before the weight computation. This is definitely important for software components, otherwise there would be so many copied or slightly modified software components when we simply collect software systems. Without proper handling of those similar components, we could not obtain reasonable and practical ranks.

We can consider another model for software components, which uses the HITS (Hyperlink Induced Topic Search) algorithm proposed for finding authorities (centers of incoming edges) and hubs (center of outgoing edges) in documents [2], [12]. We have implemented the HITS algorithm in SPARS-J and applied it to several sets of components. The initial analysis result shows that the ranking by the authority weights is similar to our component rank; however, the hub weights did not provide any useful meaning as a software component. In the WEB documents, a node with a high hub weight has many outgoing edges to other nodes, meaning that the node is a useful page pointing to other useful ones. This cannot be applicable to the software components since such hub components, which are generally event handlers or control sequence dispatchers, would be almost useless for software developers and searchers.

6.2 Software Reuse Library and Retrieval Systems

Regarding construction of software repositories and retrieval of components, much extensive research has been done. In a pioneering work by Mili at el. [17], they have investigated and classified these studies and described the characteristics of the classified methods, such as precision, recall, investment cost, pervasiveness, transparency, and so on.

SPARS-J employs a straightforward information-retrieval method to index a large amount of components effectively. This is because we think that scalability is very important for the construction of a very large software repository, which is a goal of SPARS-J, and that scalability can be accomplished with the information-retrieval method.

If we compared SPARS-J with other existing systems using the information-retrieval method, SPARS-J would be characterized by its automated process of the repository construction, low operational cost, and easiness of use. By only specifying the locations of a set of source codes, SPARS-J can perform fully automatically from the construction of a software repository to the installation of the search engine. This automated process greatly reduces the operation cost.

To facilitate its use, SPARS-J provides various features. The component rank serves as a narrowing filter for a broad query result. Even for a simple keyword list given as a query, we can easily select significant components at the top of the output list. This can be considered as a form of the specification-based browsing method proposed by Fischer [5], where time-consuming deduction computation at the retrieval phase is moved into the preprocessing phase. The use relation analysis and the component rank computation of SPARS-J would correspond to the deduction computation.

Using the pre-process analysis result, SPARS-J also provides other features, such as browsing packages by their names, chasing the use-relations, searching similar components by the similarity measure, and so on. These features help users to effectively browse and chase the relations among a large amount of components. We can say that SPARS-J is a very effective system for dealing with huge Java code assets. We are not aware of any other search system with such features.

As shown above, SPARS-J employs various analysis techniques for source code, in addition to the information-retrieval techniques for documents. We think that taking advantage of source code information is very important and essential for an effective and efficient software search system.

In [16], Maarek et al. proposed a method of searching software using the concept of lexical affinity in information retrieval and built the GURU system. Poulin and Werkmann built a search system for the Loral Federal System, in which the Structured Abstract technique is used to support retrieval [21]. Although these methods consider automatic construction of the software repositories, the scalability of those systems is unknown.

Agora [23] is a component search system for JavaBeans and CORBA using the interface information provided by the introspection mechanism. However, it does not analyze the source code, nor does it compute the significance of candidate components, as made in the component rank model.

JCentral [14] and Code Finder in Asset Locater [3] were designed as WEB-based service systems to promote asset reuse and information exchange for Java source code, applets, documents, and so on. Various searching features seem to be provided; however, deep analysis of the source code is not performed, which is essential for effective searching. SCRUPLE [19] is a prototype framework for searching code portions from the source code using a pattern description language. It provides various features of string-based pattern matching, but it has no ranking

mechanism for the matching results, which is indispensable for implementing a scalable search system.

Prospector[9] is a search engine for Java APIs. By giving a query of class paths, the system returns a list of code examples. CodeBroker [29] is an automatic code suggestion system. The system monitors and analyzes programmer's activities during an editing session, and provides code examples by using a signature matching method. We would think that these approaches apply only to the cases when we know the signatures of target classes well.

6.3 Reusability Measure

Our component rank can be considered as a true measure of software reuse. There are various research projects on software reusability, and many literatures have been already published [8], [11], [15].

Etzkorn et al.'s approach [4] is that various static metrics for C++ classes (components), such as cohesion and complexity, are measured, and these values are normalized and added. Washizaki et al.'s approach [27] is that class interface information is used to determine reusability. These approaches are based on measuring static properties of components.

Our approach does not measure the property of the components, but only uses relations among the components. We believe that the overall structure of the components represents accurately the usage history of the components, rather than the simple static metrics of the components. In our model, if a component is repeatedly reused (by not simple copying), the result rank will be higher. However, the property measures are not affected by repeated reuse.

7 Conclusion

We have proposed a novel ranking model for software components and component rank, and shown a software component search system SPARS-J based on it. Using SPARS-J, various Java programs have been analyzed, and the characteristics of the component rank have been investigated. Also, SPARS-J is currently being applied to industry.

Although the application is just the beginning, we believe that this approach shows a lot of promise for use in various situations of software development, such as searching, exploring, checking, investigating, reminding, or referring to software components, as we use dictionaries and libraries when writing a composition. SPARS-J might be considered as a Google-like system for software engineers.

In the current component rank model, we have employed an equal distribution ratio for any outgoing edge from one node. We can consider another policy such that we give various types of priority to specific outgoing edges. At this moment, we are unsure about how to determine the priority, and we think that this is a further issue.

In the component rank model, we employed only statical use relations. It is possible to extend it to use a dynamic relation such as an actual method invocation or a dynamic class binding. This approach would be more preferable since we can compute the ranks of components without their source code. The distinction between static and dynamic ranks is a very intriguing research issue to explore.

We have discussed the component rank model mainly for component search; however, we are planning to apply component rank to automatic software architecture composition. There are many literatures on structuring software architectures derived from the results of static analyses of source codes and libraries, or from the results of dynamic analyses of object code execution. We think that those analysis results could be further stabilized by the propagation and convergence computation as the component rank model.

Our component rank system is currently implemented to accept Java programs only. We will extend it to other procedural languages such as C and C++. To do this, we must define the components and use relations for those languages. Functions and procedure invocations would be practically feasible candidates, but we need further investigation and validation.

A demonstration system of SPARS-J with publicly available 180,000 components is now operating.[10] This system provides all the features discussed here, with additional and experimental features such as code-clone detection and component recommendation.

Acknowledgments

The authors express their great thanks to Daiwa Computer Co., Ltd. and Suntory Ltd. for the data collection. They also thank Hideo Nishi, Fumiaki Umemori, Kazuo Kobori, Makoto Ichii, and Eddy Parkinson of Osaka University for their contribution to the SPARS-J Project. Also, they are deeply grateful to the associate editor and anonymous reviewers for important comments and references. This project is supported by the Japan Science and Technology Corporation, Research and Development for Applying Advanced Computational Science and Technology, and also by the Japan Society for the Promotion of Science, Grant-in-Aid Scientific Research B-14380144. An early version of this paper was presented at the 25th International Conference on Software Engineering.

References

[1] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," *Computer Networks and ISDN Systems*, vol. 30, nos. 1-7, pp. 107-117, 1998.

[2] A. Chatzigeorgiou, S. Xanthos, and G. Stephanides, "Evaluating Object-Oriented Designs with Link Analysis," *Proc. 26th Int'l Conf. Software Eng. (ICSE2004)*, pp. 656-665, 2004.

[3] O. Edelstein, A. Yaeli, and G. Zodik, "eColabra: An Enterprise Collaboration & Reuse Environment," *Proc. Fourth Int'l Workshop (NGITS '99)*, pp. 229-236, 1999.

[4] L.H. Etzkorn, W.E. Huges Jr., and C.G. Davis, "Automated Reusability Quality Analysis of OO Legacy Software," *Information and Software Technology*, vol. 43, no. 5, pp. 295-308, 2001.

[5] B. Fischer, "Specification-Based Browsing of Software Component Libraries," *Automated Software Eng.*, vol. 7, no. 2, pp. 179-200, 2000.

[6] H. Fujiwara, "A New Reusability Metric Based on Reuse Results and Similarity among Programs," master's thesis, Dept. of Informatics and Mathematical Science, Osaka Univ., 2002, (in Japanese).

9. http://snobol.cs.berkeley.edu/prospector-bin/search.py.

10. SPARS-J demo page, http://demo.spars.info.

[7] H. Fujiwara, R. Yokomori, T. Yamamoto, M. Matsusita, S. Kusumoto, and K. Inoue, "Reusability Evaluation Method Using Relations among Source Code Files," *Information Processing Soc. of Japan,* technical report of Special Interest Group on Software Eng., vol. 23, pp. 155-162, 2002, (in Japanese).
[8] J. Guo and Luqi, "A Survey of Software Reuse Repositories," *Proc. Seventh IEEE Int'l Conf. and Workshop the Eng. of Computer-Based Systems,* pp. 92-100, Apr. 2000.
[9] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Component Rank: Relative Significance Rank for Software Component Search," *Proc. 25th Int'l Conf. Software Eng. (ICSE2003),* pp. 14-24, 2003.
[10] S. Isoda, "Experience Report on a Software Reuse Project: Its Structure, Activities, and Statistical Results," *Proc. 14th Int'l Conf. Software Eng.,* pp. 320-326, 1992.
[11] B. Keepence and M. Mannion, "Using Patterns to Model Variability in Product Families," *IEEE Software,* vol. 16, no. 4, pp. 102-108, 1999.
[12] J. Kleinberg, "Authoritative Sources in a Hyperlinked Environment," *J. ACM,* vol. 46, no. 5, pp. 604-632, 1999.
[13] K. Kobori, T. Yamamoto, M. Matsusita, and K. Inoue, "Classification of Java Programs in SPARS-J," *Proc. Int'l Workshop Community-Driven Evolution of Knowledge Artifacts,* 2003.
[14] R. Kraft, G. Zodik, D.A. Ford, R.Y. Pinter, D. Nicol, Q. Lu, and M. Eichstaedt, "jCentral: Search the Web for Java," *Proc. World Conf. the WWW and Internet,* pp. 626-631, 1999.
[15] C. Krueger, "Software Reuse," *ACM Computing Surveys,* vol. 24, no. 2, pp. 131-183, 1992.
[16] Y.S. Maarek, D.M. Berry, and G.E. Kaiser, "An Information Retrieval Approach for Automatically Constructing Software Libraries," *IEEE Trans. Software Eng.,* vol. 17, no. 8, pp. 800-813, 1991.
[17] A. Mili, R. Mili, and R. Mittermeir, "A Survey of Software Reuse Libraries," *Annals of Software Eng.,* vol. 5, pp. 349-414, 1998.
[18] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," technical report of Stanford Digital Library Technologies Project, 1998, http://www-db.stanford.edu/backrub/pageranksub.ps.
[19] S. Paul and A. Prakash, "A Framework for Source Code Search Using Program Patterns," *IEEE Trans. Software Eng.,* vol. 20, no. 6, pp. 463-475, 1994.
[20] G. Pinski and F. Narin, "Citation Influence for Journal Aggregates of Scientific Publications: Theory, with Application to the Literature of Physics," *Information Processing and Management,* vol. 12, no. 5, pp. 297-312, 1976.
[21] J.S. Poulin and K.J. Werkman, "Melding Structured Abstracts and World Wide Web for Retrieval of Reusable Components," *Proc. ACM sigssoft Symp. Software Reuse (SSR '95),* pp. 160-168, 1995.
[22] G. Salton, "Developments in Automatic Text Retrieval," *Science,* vol. 253, pp. 974-979, 1991.
[23] R.C. Seacord, S.A. Hissam, and K.C. Wallnau, "Agora: A Search Engine for Software Components," *IEEE Internet Computing,* vol. 6, no. 2, pp. 62-70, 1998.
[24] A. Spink, B.J. Jansen, D. Wolfram, and T. Saracevic, "From E-Sex to E-Commerce: Web Search Changes," *Computer,* vol. 35, no. 3, pp. 107-109, 2002.
[25] W.J. Stewart, *Introduction to the Numerical Solution of Markov Chains.* Princeton Univ. Press, 1995.
[26] F. Umemori, " Empirical Evaluation of Java Component Retrieval System SPARS-J," master's thesis, Graduate School of Information Science and Eng., Dept. of Computer Science, Osaka Univ., Mar. 2004, (in Japanese).
[27] H. Washizaki, H. Yamamoto, and Y. Fukazawa, "Software Component Metrics and It's Experimental Evaluation," *Proc. Int'l Symp. Empirical Software Eng. (ISESE '02),* pp. 19-20, 2002.
[28] Y.Y. Yao, "Measuring Retrieval Effectiveness Based on User Preference of Documents," *J. Am. Soc. for Information Science,* vol. 46, no. 2, pp. 133-145, 1995.
[29] Y. Ye and G. Fischer, "Supporting Reuse by Delivering Task-Relevant and Personalized Information," *Proc. 24th Int'l Conf. Software Eng. (ICSE2002),* pp. 513-523, 2002.

Katsuro Inoue received the BE, ME, and DE degrees in information and computer sciences from Osaka University, Japan, in 1979, 1981, and 1984, respectively. He was an assistant professor at the University of Hawaii at Manoa from 1984-1986. He was a research associate at Osaka University from 1984-1989, an assistant professor from 1989-1995, and a professor beginning in 1995. His interests are in various topics of software engineering such as software process modeling, program analysis, and software development environment. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Reishi Yokomori received the BE, ME, and DE degrees in information and computer science from Osaka University in 1999, 2001, and 2003, respectively. He is a researcher at Osaka University. His research interests are in the areas of program analysis and empirical software engineering. He is a member of the IPSJ, the IEICE, the IEEE, and the IEEE Computer Society.

Tetsuo Yamamoto received the BE, ME, and DE degrees in information and computer sciences from Osaka University in 1997, 1999, and 2002, respectively. He is currently an assistant professor at Ritsumeikan University. His research interests include program analysis and software reuse technology. He is a member of the IEEE and IPSJ.

Makoto Matsushita received the BE, ME, and DE degrees in information and computer sciences from Osaka University in 1993, 1995, and 1998, respectively. He is an assistant professor at Osaka University. His research interests include software process and open-source software development. He is a member of the ACM, IPSJ, and JSSST.

Shinji Kusumoto received the BE, ME, and DE degrees in information and computer sciences from Osaka University in 1988, 1990, and 1993, respectively. He is currently an associate professor at Osaka University. His research interests include software metrics, software maintenance, and software quality assurance techniques. He is a member of the IEEE, the IEEE Computer Society, IPSJ, IEICE, and JFPUG.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.