Toward a Code Search Engine Based on the State-of-Art and Practice

Vinicius C. Garcia, Eduardo S. de Almeida,
Liana B. Lisboa, Alexandre C. Martins,
Silvio R. L. Meira
Federal University of Pernambuco and
Recife Center for Advanced Studies and Systems
Recife, Pernambuco, Brazil
Email: {vcg, esa2, srlm}@cin.ufpe.br,
{liana.lisboa,alexandre.martins}@cesar.org.br

Daniel Lucrédio, Renata P. de M. Fortes
Institute of Mathematical and Computing Sciences and
São Paulo University
São Carlos, São Paulo, Brazil
Email: {lucredio, renata}@icmc.usp.br

Abstract

Software engineering and reuse-oriented tools have been studied along the last years, aiming to provide help in the software development. With the importance of reuse growing significantly, effective software reuse tools and environments started to be needed. This paper presents and discusses some works that comprise many issues related to source code search tools, covered by university and industry since 90's until today. In the end of the paper, a set of requirements is presented, integrating the features that can be found in most works of the area, serving as a basis for future work toward an effective source code search tool.

1 Introduction

Many software development organizations believe that investing in software reuse will improve their product and process productivity and quality, and are in the process of planning or developing software reuse capability.

While some organizations naively equate software reuse with a particular technology, such as object-oriented technology, in fact it has become quite clear that successful software reuse practice has much more to do with organizational management, infrastructure, and technical factors unrelated to these technologies [32]. Some of these factors (technical and non-technical) can be found in [1].

In this sense, the idea of reuse environments stands out. By integrating different factors, an environment comprises a process and its reuse-related aspects, such as assets storage, search and certification, all integrated in a set of tools that cooperate in order to offer to the software engineer a framework to perform reuse activities.

The particular case of assets storage and search may be an important way of stimulating the reuse culture in organizations trying to obtain its initial benefits [12]. Efforts must concentrate in offering subsidies and tools for the reuse of *white-box* components - where the source code is available - and already existent source code, whether from the organization itself, from previous projects, or from repositories available on the Internet. Also important is how to integrate these search functionalities in the environment, so that the reuser does not need to deal with tool integration issues.

Under such motivation, our group is currently researching software engineering and source code search tools. In [24] we present an overview of the main research works on component search and retrieval. This paper extends that work, including ideas from repository systems and software engineering environments, toward a code search tool that can be effectively used inside organizations.

2 Background

The first significant efforts in producing tightly integrated development environments were those in the area of Programming Support Environments (PSEs). As their name suggests, PSEs are collections of tools that support coding activities. Early PSEs typically provided one or more compilers, language-sensitive editors (such as syntax-directed editors), and debuggers, and sometimes other tools as well (e.g., testing or documentation utilities) [14].

PSEs comprised tightly integrated collections of tools, and as such, they were able to overcome many of the problems associated with earlier, loosely integrated environments. Their major limitation, however, is that they support only one software engineering activity - implementation - and its artifact, the source code, excluding all other major activities and their artifacts (requirements engineering, specification, among others). As became clear over

XIII ASIA PACIFIC SOFTWARE ENGINEERING CONFERENCE (APSEC'06)
0-7695-2685-3/06 $20.00 © 2006 IEEE

the years, one activity may affect another, and changes in one artifact may require changes to other related artifacts, to ensure that these remain mutually consistent. So, the identification of the need for integrated support for software engineering activities throughout the software lifecycle represents the *genesis* of Software Engineering Environments (SEE), also called CASE (Computer-Aided Software Engineering).

SEEs, like their PSE antecedents, are integrated collections of tools that facilitate software engineering activities. However, SEEs extend programming support tools with support for software engineering across the software lifecycle. These tools facilitated the development and analysis of different software artifacts. In the area of environments and tools, SEEs represent, perhaps, the most significant area of research over the past decade; they are to the full software engineering lifecycle what PSEs were to coding [14].

Through the years, a vast collection of tools have been prototyped or marketed. Some of these have been developed in the context of integrated environments, some can cooperate loosely with some others and many are freestanding. Each tool or environment is still highly specific to some context, requiring the usage of a particular language, database, compiler or integration platform [11, 14].

As software has become more pervasive and its life expectancy has increased, it has been subject to greater pressures to integrate and interact with other pieces of software, and to evolve and adapt to be used in new and unanticipated contexts, both technological (e.g., new hardware, operating systems) and sociological (e.g., new domains, business practices, processes and regulations, users) [13].

If a organization wants to achieve reuse, the software engineering environment cannot ignore these characteristics. Issues such as tool integration, flexibility, interoperability, and the incremental nature of software evolution, become critical.

3 Software Reuse Environments

According to Rine and Sonnemann [32], investment in software reuse is predictive of productivity and quality. Unfortunately, software reuse has proven to be difficult to achieve, specially when it involves changes in an organization's process and culture. One way of facilitating this aspect is to embed reuse practices and activities in the environment.

The work presented by Biggerstaff and Perlis [3] could be considered the *"first"* important work in this area. This work defines program restructuring operations (refactorings) to support the reuse of object-oriented application frameworks. The refactorings are automated by tools, and use preconditions in order to preserve the behavior of a program.

By offering tools to help in different aspects of reuse, software engineers can more easily perform reuse-related activities. For example, a process-centered environment helps developers to correctly follow reuse principles, since they are tightly integrated into the environment. An intuitive search engine stimulates the search for some piece of design or code before building it from scratch.

In this sense, many specialized technologies have been produced in both university and industry to promote particular aspects of reuse. These follow three trends: **(i)** reusable assets search engines, which are the most basic approach to promote reuse; **(ii)** repository systems, which attempt to centralize and manage all reusable information; and **(iii)** reuse environments, which attempt to cover a wider range of activities of a reuse process.

The remainder of this paper focuses on the first trend, relating university and industry experiences in this area, as well as ours, summarized in the form of a set of requirements for a code search tool.

3.1 Search Engines

An essential step in software reuse is to find previously built assets. But people often lack an idea of what they need when searching for something. They believe that a component that solves their problem may exist, but can not define neither the problem nor the solution in an adequate way.

According to Lucrédio et al. [24], the first works of the early 90's [18, 25, 29, 30] dedicate a special focus on the classification schemes used to store, and consequently retrieve, software components. An example of this concern may be seen in [30]. In this work, Prieto-Díaz proposes the utilization of a facet-based scheme to classify software components. With this scheme, it is possible to describe components according to their different characteristics, unlike the traditional hierarchical classifications, where a single node from a tree-based scheme is chosen.

However, researchers such as Maarek et al. [25] argue that the effort needed to manually classify components of a software library is too big, since these tend to grow and become huge. Moreover, the process to classify the components is susceptible subjective, so that two different people may choose different keywords or facets to describe the same asset. In this sense, Maarek et al. [25] propose the use of automatic indexing to extract, from free-text descriptors, terms or phrases that best describes a component.

In [29], Podgurski and Pierce explore component search through its execution: given a set of input values and a set of expected output values, the components are executed and those that produce the expected output are retrieved.

However, the classification scheme is only part of the component retrieval problem. The first work that focused on other aspects of the search problem was presented by Hen-

ninger [17], and was called *CodeFinder*. The *CodeFinder* uses query-construction methods that guide users through the formulation of queries, especially if users can not define their needs or do not know the terminology. Henninger, in agreement with Frakes and Pole [9], suggests that constructing queries is as important or more important than the retrieval algorithm that is used. Henninger's retrieval system combines retrieval by reformulation (which supports incremental query construction) and spreading activation (which retrieves items that do not exactly match the query, but are related to it) to help users to find information.

After this first experience with code search engines, Henninger proposes an evolutionary approach to constructing effective software reuse repositories [18]. Henninger affirms that repositories for software reuse face two interrelated problems: **(i)** acquiring the knowledge to initially construct the repository, and **(ii)** modifying the repository to meet the evolving and dynamic needs of software development organizations. In this work, Henninger outlines an approach that avoids these problems by choosing a retrieval method that utilizes minimal repository structure to effectively support the process of finding software components.

The overall advantage of Henninger's approach is that costs are incurred incrementally on an as-needed basis instead of requiring an extensive up-front repository design effort. Initially, components are only required to undergo whatever certification process is necessary to become part of a production system. Subsequent efforts can then incrementally add value to components as they are reused.

A different approach is taken by Mili et al. [28], who discuss the design and implementation of a component storage and retrieval structure based on formal specifications.

Another search engine is the *Agora* [33]. Developed by the Commercial Off-the-Shelf (COTS)-Based Systems Initiative at SEI/CMU[1], its objective was to create a worldwide, automatically generated database (repository) of software products, classified by component model. *Agora* combines introspection with Web search engines to reduce costs of bringing software components to, and finding components in, the software marketplace.

In this work, Seacord et al. talk effectively about components marketplace. However, they affirm that the combination of introspection with component search is a necessary but insufficient element of an online component marketplace. Elements required by a component marketplace that are not addressed by this work include security, electronic commerce, and quality assurance.

Therefore, the efforts of Seacord et al. toward the integration of component technology and Web search can have an impact on the emergence of on-line component marketplaces by **(i)** providing developers a worldwide distribution channel for software components, **(ii)** providing consumers a flexible search capability over a large base of available components and **(iii)** providing a basis for the emergence of value-added component qualification services, within and across specific business sectors.

Thomason et al. [35] suggest that a comprehensive component classification schema, coupled with adequate visualization and selection tools, are essential to the longevity of any component-based system. Thus they propose the *CLARiFi* approach, to develop a classification schema that identifies the properties that are important to the selection of components for a given task. This classification schema is a subset of a larger data model, which incorporates the roles of supplier, integrator, broker and certifier. This one of the first works that consider certification as an important step before storing components.

Another aspect that may be used to improve search is *context* information. From an informational perspective, context aims to provide an information space that can be *"adapted according to the user's context"*, and complemented by tools/processes that promote the socially situated construction and sharing of context. In [6], a relevant work in general information retrieval, Finkelstein et al. present a new conceptual paradigm for performing search in context, automating the search process, providing even non-professional users with highly relevant results.

This paradigm is implemented in practice in the *"IntelliZap"* system, where search is initiated from a text query marked by the user in a document he/she views, and is guided by the text that surrounds the marked query (*"the context"*). The context-driven information retrieval process involves semantic keyword extraction and clustering to automatically generate new, augmented queries, eliminating possible semantic ambiguity and vagueness.

Context information is also used by Ye and Fischer in [38], where they propose a new mechanism to find and return the results to the developer in an autonomous way. Relevant component information is customized in accordance with the knowledge and environment of the developer. They propose a process called *information delivery*, which consists in anticipating the software engineer's needs for components. The process is performed by monitoring the activities of the software engineer, such as codification or code documentation, and automatically searching for the components. The search criterion is identified inside the code. For instance, a *Javadoc* comment or a method call can be used to formulate a search criterion. However, only executable code is considered as a software component.

Washizaki and Fukazawa [37] state that conventional search techniques cannot enable prompt reuse of software because they target source code as the retrieval unit. In this sense, they propose a new component-extraction-based program search system, which analyzes a collection of Object-Oriented (OO) programs, acquires relationships among OO

[1] *Software Engineering Institute at Carnegie Mellon University*

0-7695-2685-3/06 $20.00 © 2006 IEEE

classes, and extracts reusable software components composed of some classes. The target of reuse is a program source code written in OO languages, particularly Java, and targets *JavaBeans* as a component architecture.

In [19], Holmes and Murphy propose the *Strathcona*, an Eclipse *plug-in* that finds source code examples through a search based in 6 different heuristics. The choice of the class is based on the code structure similarity that the developer is writing. After the localization in the repository, the *plug-in* relates each extracted class with the developed source code, if exists, and structure an application model of the returned class (by the search).

In [10] we present a software component search engine architecture as an Eclipse *plug-in*, called *MARACATU*. The search is performed through text mining and faceted techniques, using the *Lucene* engine [15]. The architecture is able to search Java source code and components in CVS servers, to load the selected components, to index them locally and to show the result in the Eclipse workbench. Being able to retrieve and reuse not only *"black-box"* components, but *"white-box"* components and source code too, we believe that this approach is the start point for organizations to have the first benefits of reuse, in terms of tools, since *MARACATU* is a non-intrusive way of allowing code search and retrieval.

Inoue et al. [20] propose a novel graph-representation model of a software component library. They define a *component rank model*, to be applied on a graph representation of the component library. 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 resulting rank is used to prioritize the query result so that highly ranked components are quickly seen by the user. Using the *component rank*, they developed a component search system called *SPARS-J* (Software Product Archiving and Retrieving System for Java), which treats the source files of Java classes as components.

In the same way as [10, 33], using Web search engines, *Koders* [21] was one the first *Open Source* component search engines ever developed. Koders automatically connects with different version control systems (e.g., *CVS* and *Subversion*) to identify source code, being able to recognize approximately 30 programming languages and 20 software licenses. The *Koders* official launch was in April 2005.

The first evolution of *MARACATU*, an integrated reuse environment called *ADMIRE*, has been defined and implemented by Mascena et al. [26]. According to Mascena et al., a fundamental premise for any type of reuse is the knowledge about the existence of the reusable assets. Such knowledge may already be available, for example, due to the past experience of the subject of the reuse action or may be obtained through knowledge dissemination. *ADMIRE* was based on the same concept of *information delivery* proposed by Ye and Fisher [38] in *CodeBroker*. A new reuse metric also was proposed with the goal of monitoring the reuse activities and allow the software engineer to make corrective actions across the development process. The evaluation demonstrates that the *ADMIRE environment* can be used as a basis for providing momentum at the initial stages of a reuse process [1] also acting as a supporting tool for later stages.

A second evolution of *MARACATU* was develop by Vanderlei et al. [36]. In this version, Vanderlei et al. presented the use of folksonomy concepts in a software component search engine as an alternative to improve the performance of the search engine. A set of requirements to perform component search and retrieval with folksonomy was specified and implemented, followed by the description of search using this concept in the engine. This *MARACATU* current version combines folksonomy, text-mining, and facet-based search techniques.

3.2 Discussion

Figure 1 summarizes the survey presented on the source code search tools area. According to Figure 1, there are some works (represented by an *"X"*) that mark the main changes in this research area.

In 1998, Frakes et al. [7] presented a work that has a process-centered concern. After this, other works started to think about the reuse process, such as [2, 4, 22, 23]. Afterward, with the appearance of quality models like ISO and CMM, we may notice an increasing concern about certification, metrics and component quality assurance.

Despite all these advances in different ways, *Software Reuse Environments*(SRE) are still not being widely used. In fact, an efficient SRE have not been presented yet, and the industry still waits for a good solution, which addresses the real, organization-wide issues, not only particular aspects of reuse. Although there are still some issues to be solved within particular aspects of reuse, these are not an excuse for the absence of an efficient SRE. Current technologies are good enough for building an efficient environment. Hence, next section presents the main requirements for a reuse environment, starting with a efficient source code search tool, including ideas that are present in the works covered by this survey.

4 Toward an efficient source code search tool

The time line shown in Figure 1 presents the change that is occurring since the end of the millennium, leading to the formation of an effective source code search tool, where this

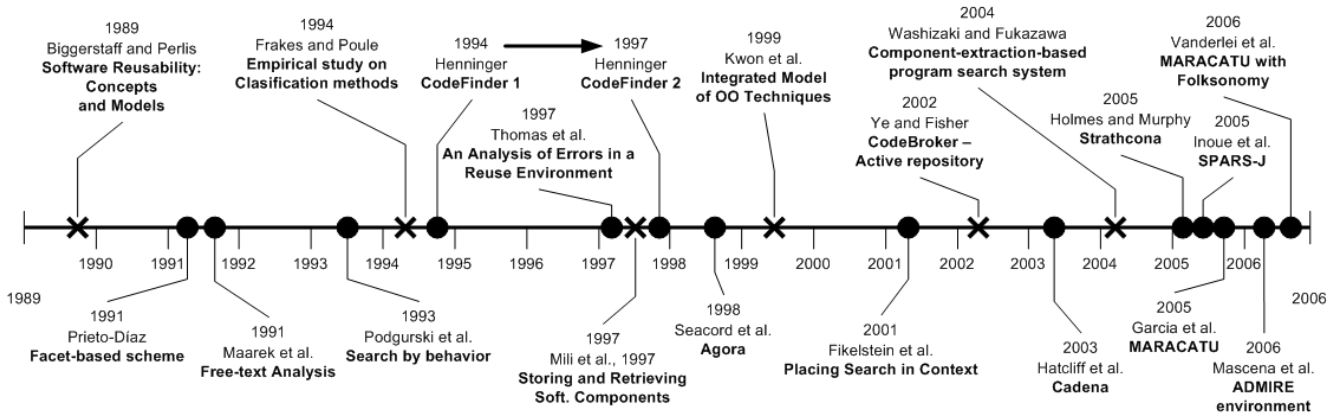

Figure 1. Research on Source Code Search Tools

particular kind of assets (source code) are made available to be reused, in different phases of the development process, in a non-intrusive way.

It also becomes evident that there are three different kinds of problems that must be dealt with when designing an effective search source code tool: **i)** efficient search and retrieval is needed, to assure that the developer is capable of finding previously built reusable assets; **ii)** a communication with different kinds of repository systems is needed, to effectively store and manage all reusable information; and **iii)** integration with an environment must be performed, dealing with the issues of using different tools during the different phases of the development process.

Substantial effort in particular aspects of each one of these issues has been already performed, as seen in this paper. Next we group all these efforts together, by presenting a set of requirements that an efficient code search tool must fulfill.

4.1 Search Engines Specific Requirements

The primary goal of the search engine is to deliver to the developer all information that he needs in order to reuse some asset, and - ideally - should not leave anything behind.

i. Retrieval algorithms. Retrieval methods can be divided into three categories: enumerated classification, faceted, and free-text indexing [8]. Through the years these were the mostly used approaches for retrieving reusable software, including most of the works presented here. Therefore, we believe that they should be part of any reuse-oriented search engine, mainly because these are already well-proven and easy-to-use solutions. Moreover, the retrieval algorithms must focus on intuitive ways to classify and identify the components with low costs. Besides, they should be based on information that is familiar to the software engineers. In this way, they can find the appropriate assets during the software development without necessarily having knowledge about the contents of the repository. In this scenario, different techniques should be experimented to address the aforementioned drawbacks [36].

ii. Active search. A search mechanism usually depends on the software engineer's initiative to start searching. However, if the SRE makes the job (search), according to the source code the software engineer is currently writing, the reuse process is incorporated to the developer's culture in a non-intrusive way [19]. Also, the chance of reuse is increased [26, 38].

iii. High recall and precision. These are the classical metrics for evaluating a search engine, and should be considered by a SRE. High precision means that most retrieved elements are relevant. High recall means that few relevant elements are left behind, without being retrieved.

iv. Query formulation. There is a natural information loss when the reuser is formulating a query. As pointed out by [17], there is also the conceptual gap between the problem and the solution, since usually components are described in terms of functionality (*"how"*), and queries are formulated in terms of the problem (*"what"*). A search engine must provide means to help the reuser to formulate the queries, consequently reducing this gap.

v. Performance. Performance is usually measured in terms of the response time. In centralized systems, the involved variables are the hardware processing power and the search algorithm complexity. In distributed scenarios, however, other variables must be considered, such as network traffic, geographical distance and the greater number of components.

vi. IDE Integration. Ideally, the search source code tool should be integrated to the developer's IDE, so that minimum overhead is required in order to use it. A flexible idea is to use *plug-in* based integration, such as in the Eclipse platform [10, 16, 19, 26, 36].

4.2 Repository Systems Requirements

Besides searching and browsing capabilities, a communication with a different kind of repositories must also provide some functionalities to manage the stored assets and all the knowledge associated to them, in order to facilitate reuse.

i. Reports. Reports are a valuable instrument, offering a general vision of how the set of reusable assets stored in the repository is being used. As examples of reports, we can cite: profile/degree of collaboration between users, profile/degree of components usage, most performed queries, most downloaded components, newest component releases, among others.

ii. Component publishing. It should be possible to publish new components, together will all information needed in order to efficiently reuse it, and the publisher's profile (author, affiliation, etc.).

iii. Component Certification. After components have been created or extracted from legacy systems, they must be evaluated and certified before stored into the repository. Quality control services must be offered, in order to increase the quality of the produced software.

iv. Users Notification. A developer should be able to register interest in some information that is maintained in the repository, so that he/she can be later notified when it changes, or when new information is added. For example, the repository can automatically notify developers when a new component is published, or when a new version of some component that he/she already reused is available.

v. Administration services. As any information system, a repository must have administrative functionalities, to allow the manager to directly manipulate data, such as users profiles, classifiers, artifact types, among others. This is important in order to solve some minor problems that might occur, and to offer some flexibility to the system.

vi. Component Manager. It should be possible to maintain different versions of one asset, allowing users to retrieve an old version, and to maintain variations from the same asset version (alternative implementations). It should also be possible to inform eventual dependencies that may exist between the stored assets.

vii. Feedback. A feedback system is important in the sense that users may register their impressions about the reused assets (in the original context or another). With this, it is possible, for example, to provide the asset developer important information about enhancements or corrections. It is also possible to implement incentive policies, rewarding the developers of the better evaluated assets.

viii. Content Indexing. All kinds of information that are maintained in the repository should be indexed, and not only the reusable assets themselves. In this way, the user can perform search in the whole repository, including other users, reports, among other kinds of information.

ix. Dynamic repository information. The repository structure should allow asset representations to be modified while users search for information, adapting itself to the changing nature of the information, and incrementally improving itself while the software engineers use it [18].

x. Security. Historically, security has been a minor issue in component search and retrieval. Since in most approaches reuse takes place in-house only, little effort is needed in order to avoid unauthorized access. However, in some scenarios this should be considered.

xi. Repository familiarity. Reuse occurs more frequently with well-known assets [30]. Therefore, a repository system should help the user to explore and get familiar with the information it maintains, so that in future searches, it is easier to locate them.

xii. Interoperability and Communication. In a scenario involving distributed repositories, which is the common surrounding environment of most organizations, it is inevitable to think about interoperability. A repository should be based on standard technologies, in order to facilitate its future expansion, integration and communication with other systems.

xiii. Referential Integrity. In SREs, reuse usually causes several software parts to reference each other. However, due to the evolutionary nature of software systems, assets are constantly added, updated and removed from their location. Thus, a SRE must guarantee that every reference has its integrity assured [23].

xiv. Software Configuration Management. Software Configuration Management can be defined as the ability to control the changes that naturally occur during the software process. The objective is to assure that the software evolves in an ordered way, reducing the confusion between the Software Engineers. This is an extensive subject, involving many tasks, such as version and modification control. In a SRE, these tasks must be performed either to control the changes in the assets as the changes in the applications [23].

4.3 Integration with an Environment Requirements

Finally, the requirements related to the integration with an environment are specified as follows.

i. Incorporate the tool in an existent development process. Harrison et al. [14] detaches the importance of semi-automated support for the software development process. We may extend this statement to the software reuse process, which should itself be treated as a piece of software - one that undergoes a similar lifecycle, including requirements specification, design, implementation, testing, analysis, etc. The adoption of the search source code tool must occur in a non-intrusive way in the software development

process.

ii. Tool integration. Development environments are composed by tools that must interoperate in order to help the software engineer through the development process. This interoperation can be achieved in different levels [23, 34]: platform, data, presentation, control and process integration. Here, this integration must also occur in a non-intrusive way, i.e. neither the tool nor the environment should need much adaptation.

iii. Technology and Language Independence. Although extensively studied, software components are still evolving, and there is no definitive solution. New component-based technologies are constantly arising. A search source code tool must maintain a basic conceptual core, independent to any technology or language, responding to the novelty without being dependent on it [23].

iv. Search in Multiple view systems. Multi-view software environments, like MVCASE [23] and Rational Rose [31], facilitate the development when different developers may have different views of the artifacts and the processes. However, it involves mapping operations and/or events from components in one view to another, and is currently an obstacle for building true multiple-view environments.

v. Reusability. In order to achieve effective reuse, the Software Engineer must be able to avoid all types of effort duplication [23]. Not just code, but every kind of asset should be reused [5] such as architecture, test cases, design, requirements, use cases, patterns, among others.

vi. Reuse Metrics. Reuse metrics [26] are used to identify the components that are highly reusable and the business areas and systems in which reuse has the high potential to provide the greatest benefits to an organization. They can identify these components that recur most frequently across the systems through domain analysis. McClure [27] describes 10 factors for reuse metrics as follows: commonality of a component, reuse threshold of a component, reuse merit of a component, reuse creation cost of a component, reuse usage costs of a component, reuse maintenance cost of a component, degree of commonality of a system, degree of reusability of a system, reuse target level and reuse merit of a system. Reuse metrics [26] are the key technology to elevating benefits from reuse together with component certification.

5 Concluding remarks

Many environments and tools have been proposed to obtain an efficient reuse. In this work, we presented a survey comprising existent software reuse tools, specifically code search tools, to increase reuse in the development process. Although we are unaware of a complete reuse environment that is suitable for real usage in the industry, we believe that current technologies are good enough for building an efficient code search tool, even though there are still issues to be resolved in the different aspects of reuse.

Most of the requirements specified in this work are the result of research in the reuse area. However, in practice most of the requirements could prove to be incomplete, or even missing. More studies and prototypes are necessary to validate the requirements presented in this survey and to construct efficient code search tool.

This study is another step in direction to an effective software reuse environment. The specification, design and implementation of this environment is being performed incrementally, through the implementation of some prototypes, like *MARACATU* [10], which is being currently developed [26, 36] by our group, as part of the RiSE[2] (Reuse in Software Engineering) reuse framework [1]. Our objective is to provide a framework for helping organizations in all aspects of implementing a reuse program.

References

[1] E. S. Almeida, A. Alvaro, D. Lucrédio, V. C. Garcia, and S. R. L. Meira. RiSE Project: Towards a Robust Framework for Software Reuse. In *IEEE International Conference on Information Reuse and Integration (IRI)*, pages 48–53, Las Vegas, USA, 2004. IEEE/CMS.

[2] A. Alvaro, D. Lucrédio, V. C. Garcia, E. S. Almeida, A. F. Prado, and L. C. Trevelin. Orion-RE: A Component-Based Software Reengineering Environment. In *WCRE 2003 - 10th Working Conference on Reverse Engineering*, pages 248–257, Victoria - British Columbia - Canada, 2003.

[3] T. J. Biggerstaff and A. J. Perlis. *Software reusability: vol. 1, concepts and models*. ACM Press, New York, NY, USA, 1989.

[4] R. M. Braga, C. Werner, and M. Mattoso. Odyssey: A Reuse Environment based on Domain Models. In *IEEE Symposium on Application-Specific Systems and Software Enginerring Technology (ASSET'99)*, pages 50–57, Richardson, Texas, 1999. IEEE/CS Press.

[5] D. D'Souza and A. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Object Technology Series. Addison-Wesley, 1999.

[6] L. Finkelstein, E. Gabrilovich, Y. Matias, E. Rivlin, Z. Solan, G. Wolfman, and E. Ruppin. Placing Search in Context: The Concept Revisited. In *Tenth International World Wide Web Conference*, Hong Kong, 2001.

[7] W. Frakes, R. Prieto-Diaz, and C. Fox. DARE: Domain analysis and reuse environment. *Annals of Software Engineering*, 5(0):125 – 141, 1998.

[8] W. B. Frakes and P. B. Gandel. Representing Reusable Software. *Information and Software Technology*, 32(10):653–664, 1990.

[9] W. B. Frakes and T. P. Pole. An Empirical Study of Representation Methods for Reusable Software Components. *IEEE Transactions on Software Engineering*, 20(8), 1994.

[2]URL: `http://www.rise.com.br`

0-7695-2685-3/06 $20.00 © 2006 IEEE

[10] V. C. Garcia, D. Lucrédio, F. A. Durão, E. C. R. Santos, E. S. d. Almeida, R. P. d. M. Fortes, and S. R. d. L. Meira. From Specification to Experimentation: A Software Component Search Engine Architecture. In I. Gorton, G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. A. Szyperski, and K. C. Wallnau, editors, *The 9th International Symposium on Component-Based Software Engineering (CBSE 2006)*, volume 4063 of *Lecture Notes in Computer Science*, pages 82–97, Mälardalen University, Västerås, Sweden, 2006. Springer-Verlag Berlin Heidelberg.

[11] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch, or Why it's hard to build systems out of existing parts. In *International Conference on Software Engineering*, 1995.

[12] M. Griss. Making Software Reuse Work at Hewlett-Packard. *IEEE Software*, 12(01):105–107, 1995.

[13] J. C. Grundy, W. B. Mugridge, and J. G. Hosking. Constructing component-based software engineering environments: issues and experiences. *Information & Software Technology*, 42(2):103–114, 2000.

[14] W. Harrison, H. Ossher, and P. Tarr. Software Engineering Tools and Environments: A Roadmap. In *The Future of Software Engineering*, pages 261–277. ACM, New York, 2000.

[15] E. Hatcher and O. Gospodnetic. *Lucene in Action*. In Action series. Manning Publications Co., Greenwich, CT, 2004.

[16] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath. Cadena: an integrated development, analysis, and verification environment for component-based systems. In *25th International Conference on Software Engineering*, pages 160–173, Portland, Oregon, 2003. IEEE Computer Society.

[17] S. Henninger. Using Iterative Refinement to Find Reusable Software. *IEEE Software*, 11(5):48–59, 1994.

[18] S. Henninger. An Evolutionary Approach to Constructing Effective Software Reuse Repositories. *ACM Transactions on Software Engineering and Methodology*, 6(2):111–140, 1997.

[19] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *27th International Conference in Software Engineering*, pages 117–125, St. Louis, MO, USA, 2005. ACM Press.

[20] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto. Ranking Significance of Software Components Based on Use Relations. *IEEE Transactions on Software Engineering*, 31(3):213–225, 2005.

[21] Koders. Koders - Source Code Search Engine, URL: http://www.koders.com, 2006.

[22] O.-C. Kwon, S.-J. Yoon, and G.-S. Shin. Component-Based Development Environment: An Integrated Model of Object-Oriented Techniques and Other Technologies, 1999.

[23] D. Lucrédio, E. S. Almeida, C. P. Bianchini, A. F. Prado, and L. C. Trevelin. Orion - A Component Based Software Engineering Environment. *JOT - Journal of Object Technology*, 3(4):51–74, 2004.

[24] D. Lucrédio, E. S. Almeida, and A. F. Prado. A Survey on Software Components Search and Retrieval. In R. Steinmetz and A. Mauthe, editors, *30th IEEE EUROMICRO Conference, Component-Based Software Engineering Track*, pages 152–159, Rennes - France, 2004. IEEE/CS Press.

[25] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An Information Retrieval Approach for Automatically Constructing Software Libraries. *IEEE Transactions on Software Engineering*, 17(8), 1991.

[26] J. C. C. P. Mascena, E. S. Almeida, V. C. Garcia, and S. R. d. L. Meira. Towards an Effective Integrated Reuse Environment. In *5Th ACM International Conference on Generative Programming and Component Engineering (GPCE)*, Portland, Oregon, USA, 2006. ACM Press.

[27] C. McClure. *Model-Driven Software Reuse*. Extended Intelligence, Inc., 1995.

[28] R. Mili, A. Mili, and R. T. Mittermeir. Storing and Retrieving Software Components : A Refinement Based System. *IEEE Transactions on Software Engineering*, 23(7), 1997.

[29] A. Podgurski and L. Pierce. Retrieving Reusable Software By Sampling Behavior. *ACM Transactions on Software Engineering and Methodology*, 2(3):286–303, 1993.

[30] R. Prieto-Díaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):88–97, 1991.

[31] T. Quatrani and G. Booch. *Visual Modeling with Rational Rose 2000 and UML*. Addison-Wesley, 1999.

[32] D. C. Rine and R. M. Sonnemann. Investiments in reusable software. A study of software reuse investiment success factors. *The Journal of Systems and Software*, 41:17–32, 1998.

[33] R. C. Seacord, S. A. Hissam, and K. C. Wallnau. Agora: A Search Engine for Software Components. Technical Report CMU/SEI–98–TR–011, ESC–TR–98–011, CMU/SEI - Carnegie Mellon University/Software Engineering Institute, August 1998. CMU/SEI - Carnegie Mellon University/Software Engineering Institute.

[34] I. Sommerville. *Software Engineering*. Pearson Education, 6 edition, 2000.

[35] S. Thomason, P. Brereton, and S. Linkman. CLARiFi: An Architecture for Component Classification and Brokerage. In *International Workshop on Component-Based Software Engineering (CBSE 2000)*, Limerick, Ireland, 2000.

[36] T. A. Vanderlei, V. C. Garcia, E. S. Almeida, and S. R. d. L. Meira. Folksonomy in a Software Component Search Engine Cooperative Classification through Shared Metadata. In *XX Brazilian Symposium on Software Engineering, Tool Session*, Florianópolis, Brazil, 2006.

[37] H. Washizaki and Y. Fukazawa. Component-Extraction-Based Search System for Object-Oriented Programs. In *Software Reuse: Methods, Techniques and Tools: 8th International Conference, ICSR 2004*, volume 3107 of *Lecture Notes in Computer Science*, pages 254–263, Madrid, Spain, 2004. Springer.

[38] Y. Ye and G. Fischer. Supporting Reuse By Delivering Task-Relevant and Personalized Information. In *ICSE 2002 - 24th International Conference on Software Engineering*, pages 513–523, Orlando, Florida, USA, 2002.

0-7695-2685-3/06 $20.00 © 2006 IEEE