From Specification to Experimentation: A Software Component Search Engine Architecture

Vinicius Cardoso Garcia[1], Daniel Lucrédio[2], Frederico Araujo Durão[1], Eduardo Cruz Reis Santos[1], Eduardo Santana de Almeida[1], Renata Pontin de Mattos Fortes[2], and Silvio Romero de Lemos Meira[1]

[1] Informatics Center – Federal University of Pernambuco & C.E.S.A.R. – Recife Center for Advanced Studies and Systems
{vinicius.garcia, frederico.durao, eduado.cruz, eduardo.almeida, silvio}@cesar.org.br
[2] Institute of Mathematical and Computing Sciences – São Paulo University
{lucredio, renata}@icmc.usp.br

Abstract. This paper presents a software component search engine, from the early specification and design steps to two experiments performed to evaluate its performance. After the experience gained from the use of this first version, several improvements were introduced. The current version of the engine combines *text mining* and *facet-based* search. The experiments indicated, so far, that using these two techniques together is better than using them separately. From the experience obtained in these experiments and in industrial tests, we point out possible improvements and future research directions, which are presented and discussed at the end of the paper.

1 Introduction

In a software development process, reuse is characterized by the use of software products in a situation that is different from when and where they were originally constructed. This idea, which is not new [1], brings crucial benefits to organizations, such as reduction in costs and time-to-market, and quality improvement.

Component repositories are among the factors that promote the success in reuse programs [2, 3]. However, the simple acquisition of a component repository does not lead to the expected benefits, since other factors must also be considered, such as management, planning, reuse processes, among others [4, 5].

Current component managers and repositories are, mostly, products that work only with *black-box* components [6], i.e., components that are packaged without the source code, inhibiting tasks such as adaptation and evolution. Moreover, the adoption of this kind of repository often implicates in reengineering the software factories, since making components available for reuse repositories (documentation, packaging) have to follow some predetermined criteria [4]. Additionally, these repositories represent isolated solutions, not associated to commonly used development tools such as Eclipse [7]. This increases the barrier for their adoption and utilization.

I. Gorton et al. (Eds.): CBSE 2006, LNCS 4063, pp. 82–97, 2006.

Thereby, an initial way of stimulating the reuse culture in organizations, and obtaining its initial benefits [8], 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.

In this context, this paper presents the specification, design and implementation of an architecture for a component search engine, to help promoting reuse during software development, and solving the mentioned problem. In previous work [9] we introduced the search engine and described our initial experience in its specification and construction. This paper makes two novel contributions:

- Some refinements on the search engine;
- An experiment that evaluates the feasibility of Maracatu search engine use in industrial contexts, aiding in the software development process with reuse of components or source code parts.

2 Basic Component Search Requirements

Current research in component search and retrieval has focused in key aspects and requirements for the component market, seeking to promote large scale reuse [9].

Lucrédio et al. [9] present a set of requirements for an efficient component search and retrieval engine, standing out:

a. High precision and recall. High precision means that most components that are retrieved are relevant. High recall means that few relevant components are left behind without being retrieved.
b. Security. In a global component market, security must be considered a primordial characteristic, since there is a higher possibility that unauthorized individuals try to access the repository.
c. Query formulation. There is a natural loss of information when users formulate queries. According to [10], there is a conceptual gap between the *problem* and the *solution*. Components are often described in terms of their functionalities, or the solution (*"how"*), and the queries are formulated in terms of the problem (*"what"*). Thus, a search engine must provide means to help the user in formulating the queries, in an attempt to reduce this gap.
d. Component description. The search engine is responsible for identifying the components that are relevant to the user, according to the query that is formulated and compared with the components descriptions.
e. Repository familiarity. Reuse occurs more frequently with well-known components [11]. However, a search engine must help the user in exploring the repository and gaining knowledge about other components that are similar to the initial target, facilitating future reuse and stimulating component vendors competition [12].
f. Interoperability. In a scenario involving distributed repositories, it is inevitable not to think about interoperability. In this sense, a search engine that functions in such scenario must be based on technologies that facilitate its future expansion and integration with other systems and repositories.

g. Performance. Performance is usually measured in terms of response time. In centralized systems, this involves variables related to the processing power and the search algorithms. In distributed systems, other variables must be considered, such as, for example, network traffic control, geographic distance and, of course, the number of available components.

These requirements, however, are related to a component market that is based on *black-box* reuse. To a search engine that also retrieves *white-box* components and reusable source code, different requirements must be considered, as presented next.

2.1 Features and Technical Requirements

A search engine based on *white-box* reuse should consider the evolving and dynamic environment that surrounds most development organizations. Differently from *black-box* reuse, where there is usually more time to encapsulate the components and to provide well-structured documentation that facilitates searching, most development repositories contain work artifacts, such as development libraries and constantly evolving components. Documentation is usually minimal, and mostly not structured.

In this sense, such engine should support two basic processes: **i)** to locate all reusable software artifacts that are stored in project repositories, and to maintain an index of these artifacts. The indexing process should be automatic, and should consider non-structured (free text) documentation; and **ii)** to allow the user to search and retrieve these artifacts, taking advantage of the index created in process **i)**.

Since in this scenario the artifacts are constantly changing, the first process must be automatically performed on the background, maintaining the indexes always updated and optimized according to a prescribed way. On the other hand, the developer is responsible for starting the second, requesting possible reusable artifacts that suits his/her problem. For the execution of these two basic processes, some macro requirements should be fulfilled:

i. Artifacts filtering. Although ideally all kinds of artifacts should be considered for reuse, an automatic mechanism depends on a certain level of quality that the artifact must have. For example, a keyword-based search requires that the artifacts contain a considerable amount of free text describing it, otherwise the engine cannot perform the keywords match. In this sense, a qualitative analysis of the artifacts must be performed, in order to eliminate low-quality artifacts that could prejudice the efficiency of the search.
ii. Repositories selection. The developer must be able to manually include the list of the repositories where to search for reusable artifacts. It must be possible, at any moment, to perform a search on these repositories in order to find newer versions of the artifacts already found, or new artifacts.
iii. Local storage. All artifacts that were found must be locally stored in a *cache*, in order to improve performance (reusable components repository centralization).

iv. Index update. Periodically, the repositories that are registered must be accessed to verify the existence of new artifacts, or newer versions of already indexed artifacts. In this case, the index must be rebuilt to include the changes.
v. Optimization. Performance is a critical issue, specially in scenarios where thousands of artifacts are stored into several repositories. Thus, optimization techniques should be adopted. A simple and practical example is to avoid to analyze and index software artifacts that were already indexed by the mechanism.
vi. Keyword search. The search can be performed through keywords usage, like most web search engines, in order to avoid the learning of a new method. Thus, the search must accept a *string* as the input, and must interpret logical operators such as *"AND"* and *"OR"*.
vii. Search results presentation. The search result must be presented in the developer's environment, so he/she can more easily reuse the artifacts into the project he is currently working on.

3 Design of the First Maracatu Search Engine

Maracatu architecture was designed to be extensible to different kinds of reusable artifacts, providing the ability to add new characteristics to the indexing, ranking, search and retrieval processes. This was achieved through the partitioning of the system into smaller elements, with well-defined responsibilities, low coupling and encapsulation of implementation details.

However, as in any software project, some design decisions had to be made, restricting the scope and the generality of the search engine. Next we discuss these decisions, and the rationale behind them:

Type of the artifacts. Although theoretically all kinds of artifacts could be indexed by the search engine, a practical implementation had to be limited to some specific kinds of artifact. This version of Maracatu is restricted to Java source code components, mainly because it is the most common kind of artifacts found, specially in open source repositories and in software factories.
CVS Repositories. Maracatu was designed to access CVS repositories, because it is the most used version control system, and also to take advantage of an already existent API to access CVS, the *Javacvs* API [13].
Keyword indexing and component ranking. To perform indexing and ranking of the artifacts, the Lucene search engine [14] was adopted. Lucene is a web search engine, used to index web pages, and it allows queries to be performed through keywords. It is open-source, fast, and easy to adapt, and this is the reason why it was chosen to become part of Maracatu architecture.
Artifacts filtering. As a strategy for filtering the *"quality"* artifacts (with high reuse potential), the *JavaNCSS* [15] was used, to perform source code analysis in search for JavaDoc density. Only components, with more than 70% of its code documented, are considered. This simple strategy is enough to guarantee that Lucene is able to index the components, and also requires little effort to implement.

User environment. Maracatu User Interface, where the developer can formulate the queries and view the results, was integrated to Eclipse platform, as a *plug-in*, so that the user does not need to use a different tool to search the repositories.

Maracatu architecture is based on the client-server model, and uses *Web Services* technology [16] for message exchange between the subsystems. This implementation strategy allows Maracatu Service to be available anywhere on the Internet, or even on corporative Intranet, in scenarios where the components are proprietary.

Maracatu is composed of two subsystems:

Maracatu Service: This subsystem is a Web Service, responsible for indexing the components, in background, and responding to user's queries. It is composed of the following modules: the **CVS** module, which accesses the repositories in the search for reusable components; the **Analyzer**, responsible for analyzing the code in order to determine if it is suitable for indexing; the **Indexer**, responsible for indexing the Java files that passed through the Analyzer, also rebuilding the indexes when components are modified or inserted; the **Download** module, which helps the download (check-out) process, when the source code is transferred to the developer machine, after a request; and the **Search** module, which receives the parameters of a query, interprets it (for example, *"AND"* and *"OR"* operators), searches the index, and returns a set of index entries.

Eclipse *plug-in*: This subsystem is the visual interface the developer sees. It acts as a Web Service client to access Maracatu Service.

The first version of Maracatu can be seen in Figure 1, which shows Maracatu plug-in[1] being used in Eclipse development environment **(1)**.

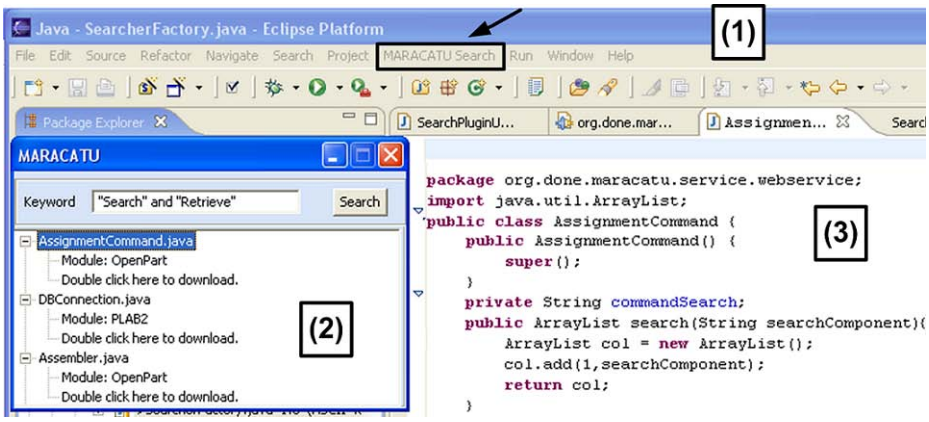

Fig. 1. Maracatu running on Eclipse environment

[1] The version 1.0 of the plug-in may be obtained on the project site `http://done.dev.java.net`

The Figure shows a screen of the plug-in **(2)**, where the developer may type a string to retrieve the components. In the example, a search was performed with the string *"Search" & "Retrieve"*, obtaining as a result the following components: *AssignmentCommand*, *DBConnection*, *Assembler*, among others. From this result, it is possible to identify which project (repository) this component belongs to (represented in *"Module"*), and download the component to the local machine. Next, the developer may import the component into his/her Eclipse project **(3)**. In the example of the Figure, the developer has chosen the *AssignmentCommand*.

The first version of Maracatu plug-in implementation contained 32 classes, divided into 17 packages, with 955 lines of code (not counting comments).

4 Maracatu Search Engine: Current Stage

After the first release of Maracatu, and its initial utilization in the industry, several enhancements started to be suggested by its users. Some of these were added, giving origin to the current version of the tool. Maracatu's first version was used to aid in the second version development. It helped the team to understand how to use some API, consulting the open source code as example of its use and to reduce the time to release the second prototype.

Next sections describe the new features that were included. The improvements took place both in the client (plug-in) and in the server side (Maracatu Service).

4.1 Usability Issues

As expected, the first problems detected by the users were related to the User Interface. In this sense, improvements were introduced into Maracatu's Eclipse *plug-in*:

i) Component pre-visualization: Before actually downloading a component, it is interesting to have a glimpse on its content, so that the user may determine if it is worth to retrieve that component or not. This saves considerable time, since the *check-out* procedure, needed to download a component from CVS, requires some processing. In this sense, two options were implemented, as shows Figure 2. The user may choose, in a pop-up menu **(1)**, either to see a text **(2)** or UML **(3)** version of the component, which he/she can then analyze before actually downloading the component. The UML was obtained by a parser which analyze the Java code and perform a transformation to write the UML.

ii) Drag and Drop functionality: With this new feature, components listed in the tree view can be directly dragged to the user workspace project, been automatically added to the project.

iii) Server Configuration: In the first version of Maracatu, the repositories addresses were not dynamically configurable. The user could not, for example, add new repositories without manually editing the server's configuration files. In order to solve this inconvenience, a menu item was added, to show a window where the user can configure which repositories are to be considered in the search.

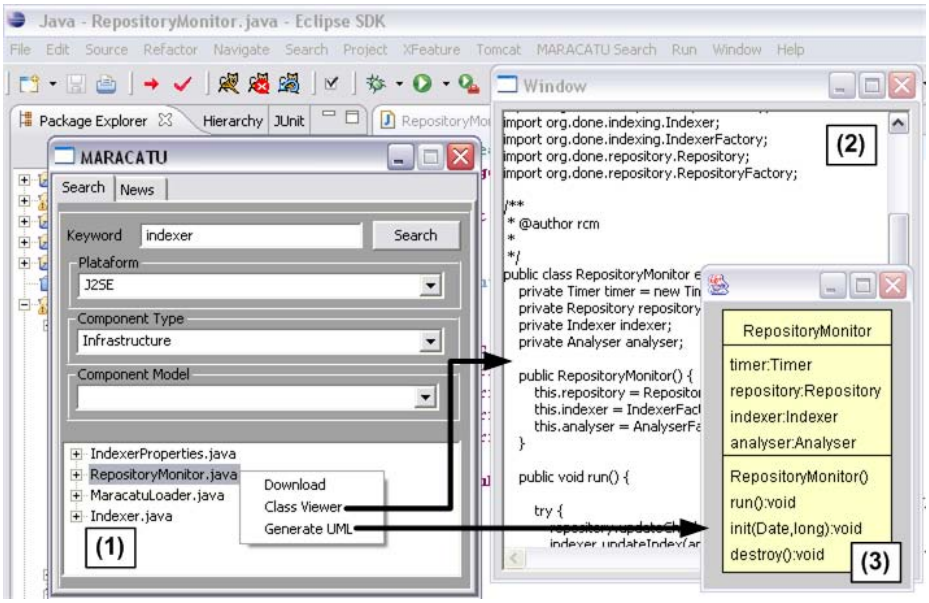

Fig. 2. Class Viewer and UML Generator

4.2 Facet-Based Search

The current version of Maracatu supports Facet-Based classification and search [17] of the components. Components now can be browsed by platform, component type and component model. It is also possible to combine facet-based search with text-based search. By choosing only the desired facets, the search universe is reduced, improving the search performance.

A new module, called Classifier, was introduced in the server-side of Maracatu's architecture. This module is responsible for:

i) Reading the components from Maracatu's repository, identifying the facets to be extracted and inserted in the new search method. The extractor looks for pre-defined facets, defined in a configuration file, together with rules for their identification. Currently the rules consist of a combination of correlated terms that must appear inside a component's code in order to determine if it is classified within the facet. New facets can be inserted by modifying this configuration file. **ii)** After the identification and extraction of the facets, components are classified according to them. The extraction and classification works together.

In the client side (Eclipse *plug-in*), modifications were made on the interface, with a *"selector"* for each facet, allowing the developer to select the desirable values for each one. The field for typing the regular text-based query was maintained, so the user may combine facet-based search with text-based search. The search is now preceded by a filtering, which excludes components that do not satisfy the constraints (facets). The keyword-based search is then performed over the filtered result.

Currently, Maracatu is capable of classifying components according to three facets (F), with the following values:

F1: Platform - **Values:** J2EE, J2ME or J2SE;
F2: Component Type - **Values:** Web Services, GUI, Networking, Infrastructure, Arithmetic, Security, Java 3D or Data Source; and
F3: Component Model - **Values:** EJB, CORBA or JavaBeans.

The user may choose combinations of these facets and values, performing queries such as: retrieve all Infrastructure or Networking components that are developed for J2EE Platform in the EJB Component Model.

5 Practical Usage in the Industry

Currently, the second version of Maracatu is being used in the industrial context, at C.E.S.A.R.[2], a Brazilian company. It is initially being used in two projects, developed by RiSE[3] group. These projects involve the development of a component manager and a shared component library for Brazilian companies. The two projects are supported by the Brazilian Government, under a budget of around $1.5 millions. The team that uses Maracatu in these projects is composed by 13 members, divided as follows: project manager (1), software quality engineer (1), software configuration manager (1), team leader (1), technical leader (1) and software engineers (8). The experience gained in this usage is important to identify opportunities for new features and improvements.

These projects' repository contains 5 sub-projects, involving around 4200 artifacts created and shared by the development team. These artifacts may be reused in different ways, offering different kinds of contribution to new projects: source code can be directly reused, but they can also serve as examples of some particular implementation or structural design.

The second version of Maracatu plug-in implementation contained 106 classes, divided into 55 packages, with 3844 lines of code (not counting comments).

6 Experiments

Two experiments were performed in order to analyze and compare the mechanisms of keyword matching and facet searching. The goal was to verify if the second version became more useful than the first one, since the facet mechanism was included.

For each experiment, four metrics were considered: the *recall*, the *precision* and the *f-measure*. Recall is the number of relevant components retrieved over the number of relevant components in the database [18]. The precision is the number of relevant components retrieved over the total number of components retrieved. Recall and precision are the classic measures of the effectiveness of

[2] Currently, this company has about 700 employees and is preparing to obtain CMMi level 3.
[3] `http://www.cin.ufpe.br/~rise`

an information retrieval system. Ideally, a search mechanism should have good precision and good recall. To assess this, mechanisms can be evaluated through the *f-measure*, which is the harmonic mean of precision and recall [19]. The closer the *f-measure* is to 1.0, the better the mechanism is. But this will only occur if both precision and recall are high. If some mechanism have excellent precision, but low recall, or excellent recall, but low precision, the *f-measure* will be closer to zero, indicating that this mechanism does not perform well in one of these criteria.

6.1 Context

According to Prieto-Díaz [17] the facet approach provides higher accuracy and flexibility in classification. The facet searching is based on the controlled vocabulary and relies on a predefined set of keywords used as indexing terms. These keywords are defined by experts and are designed to best describe or represent concepts that are relevant to the domain question.

From these experiments, we expect to obtain similar results, i.e., the facet approach should have better accuracy in classifying the components, and therefore the recall should be higher. On the other hand, free text search should have higher precision, since it only retrieves components that has terms provided in the query. If our results are correct, the combination of text and facet-based search should provide the best results, resulting in higher *f-measure* than the isolated approaches. These results would indicate that Maracatu's mechanisms were consistently implemented, and that the theory behind it is well-founded.

We considered that values close to 50 % for recall and values close to 20 % for precision are satisfactory, since they come close to measurements made by other authors [20, 11]. However, these values are only considered as a reference, and these results were not included in the hypotheses of the experiments.

The dependent variables for the experiments are recall, precision, search time, and *f-measure*. The independent variable is the searching method with three approaches: keyword, facet, and keyword + facet. Differences in subjects' skills were also considered, to explain the results.

The null hypotheses, i.e., the hypotheses that the experimenter wants to reject, are:

- H_{0a}: facet-based search has lower recall than keyword search
- H_{0b}: keyword-based search has lower precision than facet-based search
- H_{0c}: the combination of facet-based and keyword-based search does not have a greater *f-measure* than the isolated approaches

By rejecting these hypotheses, we expect to favor the following alternative hypotheses:

- H_1: facet-based search has higher recall than keyword search
- H_2: keyword-based search has higher precision than facet-based search
- H_3: the combination of facet-based and keyword-based search have a greater *f-measure* than the isolated approaches

If null hypotheses H_{0a} and H_{0b} are rejected, the results would indicate the theory that facet-based search retrieves more relevant components, and that keyword-based search is more precise. But the main result to be expected comes from null hypothesis H_{0c}. If rejected, the results would indicate that the combination of facet-based and keyword-based search takes advantage of the best of each approach, producing a better overall result. By following this rationale, the new version of Maracatu is more useful than the first one.

6.2 Preparation of the Experiments

In the first experimental environment, a repository was divided into 14 index files for 4017 source code components distributed in 8 different projects from Java.net (`http://java.net/`) and SourceForge (`http://www.sourceforge.com`) developers site, and two RiSE projects. The second experimental environment had a repository divided into 14 index files for 3819 source code components distributed in 7 different projects, from the same developers site.

One particularly challenging task is to obtain a precise measure of the recall, since the experimenter needs to know exactly how many relevant components exist in the repository for each query. To overcome this problem, both experiments adopted the same strategy: one of the projects inserted into the repository, called **known project**, (with about 200 components), was from a very specific domain, and was very well known by an expert. In this way, he could provide a number of relevant components for each query with some assurance, since he has a good knowledge of that project. Each experiment had a different known project.

The experiments were conducted in a single machine, a Compaq Presario with 2,4 GHz, 512 MB RAM and Windows XP SP1. The subjects in this study were 4 researches of the RiSE Group and C.E.S.A.R, primarily software engineers and analysts. Each subject was given a set of ten queries for each searching method (keywords, facets and keywords + facets), and was asked to find all items in the repository relevant to the query. The expert for each known project should be consulted in this activity.

The queries were elaborated with the help of the expert for each known project, and were specific to its domain, so that the number of relevant components outside the known project - which would be unknown to the expert - would be minimum.

6.3 Analysis of Experimental Results

Recall. Table 1 shows the recall results for both experiments. For each approach, the table shows the mean of the recall for the ten queries, the standard deviance and the variance.

In experiment 2, if we consider the worst case of the standard deviance, the null hypothesis H_{0a} - facet-based search has lower recall than keyword search- fails to be rejected, since there is a possibility that keyword-based approach has greater recall than facet-based. However, in experiment 1, even considering

Table 1. Recall for both experiments

Approach	Experiment 1			Experiment 2		
	Recall	Std.Dev.	Variance	Recall	Std.Dev.	Variance
Keyword	0,4356	0,1434	0,0206	0,4867	0,2813	0,0791
Facet	0,8046	0,1562	0,0244	0,6936	0,2749	0,0756
Kw./Facet	0,4584	0,1646	0,0271	0,3158	0,2665	0,0710

the worst case of the standard deviance, the null hypothesis H_{0a} is rejected. This favors alternative hypothesis H_1: facet-based search has higher recall than keyword search.

Precision. Table 2 shows the precision results for both experiments. For each approach, the table shows the mean of the precision for the ten queries, the standard deviance and the variance.

Table 2. Precision for both experiments

Approach	Experiment 1			Experiment 2		
	Precision	Std.Dev.	Variance	Precision	Std.Dev.	Variance
Keyword	0,2084	0,2745	0,0753	0,1556	0,2655	0,0705
Facet	0,0071	0,0102	0,0001	0,0155	0,0238	0,0006
Kw./Facet	0,2616	0,2786	0,0776	0,2530	0,4658	0,2169

In both experiments, if looking only at the mean values, one may tend to think that null hypothesis H_{0b} - keyword-based search has lower precision than facet-based search - was rejected. However, although this is probably true, it is not guaranteed by statistical results, since the standard deviance was too high, which may indicate that the mean could drastically change. However, in practice, due to the high difference in the mean values in both experiments, we can favor the alternative hypothesis H_2: keyword-based search has higher precision than facet-based search.

f-measure. Table 3 shows the *f-measure* results for both experiments. For each approach, the table shows the mean of the *f-measure* for the ten queries, the standard deviance and the variance.

By looking at these results, we may immediately discard the facet-based approach, since it has a very low *f-measure* for both experiments. However, null hypothesis H_{0c} - the combination of facet-based and keyword-based search does not have a greater *f-measure* than the isolated approaches - cannot be statistically rejected by these results. If we look at both experiments, and if we consider the worst case of the standard deviance, the mean could change drastically, and the *f-measure* for the keyword approach could be higher than the keyword + facet approach.

Table 3. *F-measure* for both experiments

Approach	Experiment 1			Experiment 2		
	F-meas.	Std.Dev.	Variance	F-meas.	Std.Dev.	Variance
Keyword	0,2544	0,2584	0,0668	0,2109	0,3181	0,1012
Facet	0,0136	0,0189	0,0004	0,0294	0,0443	0,0020
Kw./Facet	0,3127	0,2592	0,0672	0,2361	0,2559	0,0655

However, in practice, considering just the mean values, both experiments tend to reject Null hypothesis H_{0c}, since in both cases the combination of facets and keywords had a greater *f-measure*. Thus, if we had to make a decision, we would favor alternative hypothesis H_3: the combination of facet-based and keyword-based search have a greater *f-measure* than the isolated approaches. However, more experiments are needed in order to provide a more solid confirmation of this hypothesis.

6.4 Discussion

Subject preferences for the searching methods was obtained by asking the subjects to answer which approach was preferred. Keyword + facet was ranked higher, followed by keyword and only then the facets.

The three null hypotheses were practically rejected, although not statistically. This favors the alternative hypotheses, and specially H_3, which states that the new version of Maracatu, combining facet-based search with keyword-based search, is more useful than the first one, which only had keyword-based search.

As expected, the recall and precision rates, in the best cases, were very close to the values obtained by other authors [20] [11] (50% recall and 20% for precision). We can not say which mechanism is better, nor that these mechanisms are similar, since several other factors could influence the result. The same set of components and queries should be replicated to all mechanisms in order to obtain a more meaningful comparison result. However, this indicates that the research on Maracatu is on the right direction.

7 Related Work

The Agora [21] is a prototype developed by the SEI/CMU[4]. The objective of the Agora system is to create a database (repository), automatically generated, indexed and available on the Internet, of software products assorted by component type (e.g. *JavaBeans* or *ActiveX* controls). The Agora combines introspection techniques with Web search mechanisms in order to reduce the costs of locating and retrieving software components from a component market.

The Koders [22] connects directly with version control systems (like CVS and Subversion) in order to identify the source code, being able to recognize

[4] Software Engineering Institute at Carnegie Mellon University.

30 different programming languages and 20 software licenses. Differently from Maracatu, which can be used in an Intranet, Koders can be only used via its Web Site, which makes it unattractive for companies that want to promote in-house reuse only, without making their repositories public.

In [23], Holmes and Murphy present *Strathcona*, an Eclipse plug-in that locates samples of source code in order to help developers in the codification process. The samples are extracted from repositories through six different heuristics. The *Strathcona*, differently from Maracatu, is not a Web Service client, and thus it is not as scalable as Maracatu. Besides, Maracatu can access different remotely distributed repositories, while the *Strathcona* can access only local repositories.

Another important research work is the *CodeBroker* [11], a mechanism for locating components in an active way, according to the developer's knowledge and environment. Empirical evaluations have shown that this kind of strategy is effective in promoting reuse. From the functional view, Maracatu follows the same approach as *CodeBroker*, except for being passive instead of active.

8 Maracatu's Agenda for Research and Development

As a result of the research and tests made with the tool, the team responsible for the project identified the necessity for the development of new features and new directions for research. A formal schedule of these requirements is being defined by C.E.S.A.R. and RiSE group, and will address the following issues.

8.1 Non-functional Requirements

Usability. Macaratu's usability might be enhanced with features such as giving the user the possibility to graphically view the assets and its interdependencies. This would help the user to keep track of assets cohesion and to learn more about the assets relationships and dependencies. Another usability feature could be to highlight the searched text. And finally, it would be interesting for the user to select the repositories he/she wants to search, as an additional filter.

Scalability. On the server side, there are not features for load balancing. This will be an important feature in the future, as the tool starts to be used with a larger number of developers simultaneously searching for assets on the Intranet or even on the Internet.

Security. A common problem that a company may face when promoting reuse is the unauthorized access to restricted code. The idea is to improve software reuse, but there are cases where not every user can access every artifact. User authentication and authorization need to be implemented in order to solve these questions.

8.2 Functional Requirements

Improved facet search. The facet search might be enhanced, by using more complex, flexible and dynamic rules. Currently, facet rules are specific for Java source code, and use a very simple structure. A rule engine should be used to

improve it. This would bring the necessary flexibility for the administrator or the developer to define specific semantic-aware rules to associate pre-defined facets. Besides, a more flexible facet extractor would be easier to adapt to organizational structures, facilitating the adoption of the search engine.

Semantic Search. Semantic search might be added to improve recall, since it would retrieve not only the specific assets the user searched for, but also others that are semantically related. Current facet search is a form of semantic search, since the facets are semantically defined to represent and group some information on the repository. However, new semantic engines could provide more benefits.

Specialized Algorithm. On its second prototype, Maracatu uses the Lucene Search system to index and retrieve source code. This algorithm is not optimized or specialized for source code search. A feature that might be added is to count the source code dependencies when indexing and ranking it. So a developer could choose to retrieve the assets with less dependencies, for example. One example of such work can be seen on Component Rank [24].

Metrics. The use of more complex metrics than JavaNCSS might be interesting. Currently the only metric information used is the amount of Javadoc documentation. We can evaluate other code quality features in order to improve the filter process.

Query by reformulation. There is a natural information loss when the reuser is formulating a query. As pointed out by [10], 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"*). In [11], the authors state that retrieval by reformulation *"is the process that allows users to incrementally improve their query after they have familiarized themselves with the information space by evaluating previous retrieval results."*.

Information Delivery. Most tools expect user's initiative to start searching for reusable assets. Unfortunately, this creates a search gap, because the user will only search for components he/she knows or believes to exist in the repository [11]. On the other hand, using context-aware features, the tool can automatically search for relevant information without being requested, bringing components that the user would not even start looking for, increasing the chance of reuse.

We are aware that this is not a definitive set of improvements. However, these are proved solutions that could increase Maracatu's performance and usefulness.

9 Concluding Remarks

Since 1968 [1], when McIlroy proposed the initial idea of a software component industry, the matter has been the subject of research. Over from decades [9], the component search and retrieval area evolved, with mechanisms that, initially, facilitated the reuse of mathematical routines, up to robust mechanisms, which help in the selection and retrieval of *black-box* components, either in-house or in a global market.

In this paper, we presented Maracatu, a search engine for retrieving source code components from development repositories. The tool is structured in a client-sever architecture: the client side is a *plug-in* for Eclipse IDE, while the server side is represented by a web application responsible for accessing the repositories in the Internet or Intranets. Two versions of the engine were developed so far, with new features being added as it is used in industrial practise. We also presented two experiments, comparing the text matching mechanism (first version) with the facet mechanism implemented in the last version. The experiment showed that the facet-based mechanism alone does not have good performance but, when combined with text-based search, is a better overall solution.

Additionally, we discussed Maracatu's agenda for future research and development, listing it features still to be implemented. Issues concerned with usability, scalability and security gain importance in future releases, as pointed out by the experiments and pactical usage. Particularly, the facet searching mechanism could benefit from more sophisticated, flexible and dynamic rules. Semantic search would be another important approach to be studied, as well as more specialized algorithms for component ranking.

In the view of the RiSE framework for software reuse [25], Maracatu is a search tool to incorporate the first principles and benefits of reuse into an organization. However, reusability will not occur by itself, and it is an illusion to think that the adoption of tools could do it either. There must be a strong organizational commitment to reuse program; adherence to a reuse process; an effective management structure to operate a reusability program with the resources and authority required to provide the overall culture to foster reuse. Maracatu facilitates the task of reusing software artifacts, but we hope that the first benefit it brings can encourage project managers and CIOs to pay attention to the software reuse as a viable and mandatory investment in their software development agenda.

References

1. McIlroy, M.D.: Software Engineering: Report on a conference sponsored by the NATO Science Committee. In: NATO Software Engineering Conference, NATO Scientific Affairs Division (1968) 138–155
2. Frakes, W.B., Isoda, S.: Success Factors of Systematic Software Reuse. IEEE Software **11**(01) (1994) 14–19
3. Rine, D.: Success factors for software reuse that are applicable across Domains and businesses. In: ACM Symposium on Applied Computing, San Jose, California, USA, ACM Press (1997) 182–186
4. Morisio, M., Ezran, M., Tully, C.: Success and Failure Factors in Software Reuse. IEEE Transactions on Software Engineering **28**(04) (2002) 340–357
5. Ravichandran, T., Rothenberger, M.A.: Software Reuse Strategies and Component Markets. Communications of the ACM **46**(8) (2003) 109–114
6. Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming. Addison Wesley (2002)
7. Gallardo, D., Burnette, E., McGovern, R.: Eclipse in Action. A Guide for Java Developers. In Action Series. Manning Publications Co., Greenwich, CT (2003)

8. Griss, M.: Making Software Reuse Work at Hewlett-Packard. IEEE Software **12**(01) (1995) 105–107
9. Lucrédio, D., Almeida, E.S., Prado, A.F.: A Survey on Software Components Search and Retrieval. In Steinmetz, R., Mauthe, A., eds.: 30th IEEE EUROMICRO Conference, Component-Based Software Engineering Track, Rennes - France, IEEE/CS Press (2004) 152–159
10. Henninger, S.: Using Iterative Refinement to Find Reusable Software. IEEE Software **11**(5) (1994) 48–59
11. Ye, Y., Fischer, G.: Supporting Reuse By Delivering Task-Relevant and Personalized Information. In: ICSE 2002 - 24th International Conference on Software Engineering, Orlando, Florida, USA (2002) 513–523
12. Banker, R.D., Kauffman, R.J., Zweig, D.: Repository Evaluation of Software Reuse. IEEE Transactions on Software Engineering **19**(4) (1993) 379–389
13. NetBeans: Javacvs project (2005)
14. Hatcher, E., Gospodnetic, O.: Lucene in Action. In Action series. Manning Publications Co., Greenwich, CT (2004)
15. Lee, C.: JavaNCSS - A Source Measurement Suite for Java (2005)
16. Stal, M.: Web services: beyond component-based computing. Communications of ACM **45**(10) (2002) 71–76
17. Prieto-Díaz, R.: Implementing faceted classification for software reuse. Communications of the ACM **34**(5) (1991) 88–97
18. Grossman, D.A., Frieder, O.: Information Retrieval. Algoritms and Heuristics. Second edn. Springer, Dordrecht, Netherlands (2004)
19. Robin, J., Ramalho, F.: Can Ontologies Improve Web Search Engine Effectiveness Before the Advent of the Semantic Web? In Laender, A.H.F., ed.: XVIII Brazilian Symposium on Databases, Manaus, Amazonas, Brazil, UFAM (2003) 157–169
20. Frakes, W.B., Pole, T.P.: An Empirical Study of Representation Methods for Reusable Software Components. IEEE Transactions on Software Engineering **20**(8) (1994)
21. Seacord, R.C., Hissam, S.A., Wallnau, K.C.: 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 (1998) CMU/SEI - Carnegie Mellon University/Software Engineering Institute.
22. Koders: Koders - Source Code Search Engine, URL: http://www.koders.com (2006)
23. Holmes, R., Murphy, G.C.: Using structural context to recommend source code examples. In: 27th International Conference in Software Engineering, St. Louis, MO, USA, ACM Press (2005) 117–125
24. Inoue, K., Yokomori, R., Fujiwara, H., Yamamoto, T., Matsushita, M., Kusumoto, S.: Component Rank: Relative Significance Rank for Software Component Search. In: 25th International Conference on Software Engineering (ICSE2003). (2003) 14–24
25. Almeida, E.S., Alvaro, A., Lucrédio, D., Garcia, V.C., Meira, S.R.L.: RiSE Project: Towards a Robust Framework for Software Reuse. In: IEEE International Conference on Information Reuse and Integration (IRI), Las Vegas, USA, IEEE/CMS (2004) 48–53