Evaluating the Efficiency of Retrieval Methods for Component Repositories

Oliver Hummel, Werner Janjic & Colin Atkinson
Chair of Software Technology, University of Mannheim
{hummel, wjanjic, atkinson}@informatik.uni-mannheim.de

Abstract

Component-based software reuse has long been seen as a means of improving the efficiency of software development projects and the resulting quality of software systems. However, in practice it has proven difficult to set up and maintain viable software repositories and provide effective mechanisms for retrieving components and services from them. Although the literature contains a comprehensive collection of retrieval methods, to date there have been few evaluations of their relative efficiency. Moreover, those that are available only study small repositories of about a few hundred components. Since today's internet-based repositories are many orders of magnitude larger they require much higher search precision to deliver usable results. In this paper we present an evaluation of well known component retrieval techniques in the context of modern component repositories available on the World Wide Web.

1. Introduction

The basic motivation for software reuse is simple. It is about "*creating software systems from existing software rather than building software systems from scratch"* as Krueger [1] put it in 1992. It is expected to reduce the time needed to developed software applications and improve the quality of delivered software products. Krueger's definition includes the potential reuse of any kind of asset during a software development process. Approaches like product line engineering [3] and design patterns [4] have proven particularly successful in this regard. However, the original vision presented by McIlroy [5] in 1968 focussed on the reuse of software components obtained from so-called component markets. The availability of components has clearly grown over the years but component markets have yet to make a significant impact on practical software development. This is for example demonstrated by the limited number of component vending sites available on the Internet and last year's surprising shutdown of the Universal Business Registry for web services [6].

Some successful "in-house" implementations of the component market concept have been reported by companies like IBM [9] or GTE [10] in the past. However, the size of their component repositories was limited to a few hundred components - the same approximate size as the component repository prototypes that were investigated in the 1990s (e.g. [7], [8]). Some experts like Poulin [19] have identified a size of about 200 components as the upper limit for manually maintained component repositories since the content of larger repositories tends to degenerate (the UBR is a good example for this hypothesis). However, researchers have long tried to exceed this limit and automatically populated repositories [20] with more than one hundred thousand assets have been reported more recently [2]. One rationale for this approach is to try to improve the chances of finding a suitable component. Another is the enormous size of publicly available class and function libraries, which often exceed thousands of components, and company version control systems which often contain hundreds of thousands of components. It is obvious that only automatically populated repositories will be able to cope with the numbers of reusable assets available today. The reuse of open source software from the web presents even more challenges to the reuse community and there has been general pessimism in some quarters that the so-called component storage and retrieval problem will be solved in the near future [13] [16].

Very recently however, some commercial search engines focusing on (open) source code from the Internet have demonstrated that repositories with millions of components are now technically feasible. The four major players in this segment are, in the order of appearance on the market: Koders.com, Krugle.com, merobase.com and Google's new code search engine (google.com/ codesearch). We believe that repositories of this size require a shift of priorities in component retrieval research. Where the main problem used to be to find any kind of matching (or at least a similar) component in a small repository, today the problem has shifted to finding the "best" matching component from a variety of candidates. This can only be achieved by improved retrieval techniques. Because of the previous lack of real world reuse repositories there are only a few publications examining the efficiency of these techniques. Even the authors of an often cited survey [16] admitted that their data about retrieval efficiency was largely estimated from common sense and the few meager studies accessible to them at the time. Even worse, the few viable results that are obtainable (e.g. [7]) are all based on repositories with far less than one thousand components. It is questionable whether these results will be scaleable for larger collections and will hold for repositories with millions of assets. The following small example illustrates why.

Consider the interface of a simple stack component that might have the following form in Java:

```
public class Stack {
  public void push(Object o) {}
  public Object pop() {}
  public int size() {}
}
```

Using signature matching [15], a classic and well known retrieval technique the component could be represented as follows:

```
Object -> void
void -> Object
void -> int
```

In a collection of about 1000 components such a signature might appear less then ten times and a stack could be identified with relative ease amongst them. However, performing the same query with the data pool of the merobase search engine (containing almost 10 million components) delivers more than 40.000 results.
Given these new developments it has become apparent that new analyses of the effectiveness of classical retrieval techniques, and combinations thereof, are required to improve the theoretical foundation for component repositories. In section 2 we provide a more detailed overview of the state of the art in component-based software reuse, component retrieval techniques and known evaluations. In section 3 we present our own evaluation of retrieval techniques based on modern code search engines from the web and discuss the results in section 4. Finally, we summarize and conclude our contribution in section 5.

2. Component Retrieval Background

Many attempts have been made to solve the problem of effectively storing large numbers of potential reuse candidates in a component repository. However, not only has this so-called "repository problem" [13] proven hard to solve, the question of how a component should best be represented (called the "reuse representation problem" by [7]) has also been a stumbling block. Given that the existing component repositories on the web are almost all source code centric and offer only basic text-search capabilities (see next subsection) it is difficult to use them for more sophisticated reuse approaches than just "code scavenging" [1]. This practice describes the copying and pasting of small code snippets into the system under development and is discouraged in many publications such as the Anti-pattern book [21]. It requires a lot of effort to find appropriate snippets and their use is more likely to degenerate the design of the system under development than to improve its quality. Although software reuse has been the subject of research for almost four decades, there is still no clear picture of when and how components should be used in a development process and how they should be stored in a repository. Even modern development methodologies contain only very abstract guidelines to select components based on their interface. Kratz et. al. [22] have recently shown that there is indeed some relation between the interface and the functionality of a component. However, since candidates might not match perfectly, a feedback loop may be necessary in which either the design or the candidate have to be adapted. This idea is best described in [11] so far.
Our own experience with reuse repositories indicates that the best kind of component search to use in a development process depends on the point of time at which the search is performed rather than on the nature of the process itself. The earlier reusable components are searched during a system's development the less design work is likely to have been carried out. Hence a general text-based search is more useful in early development phases and can feed back valuable information about potential components and their interfaces into the design process. On the other hand, if component search is carried out at a relatively late point in the development process an interface-driven search approach is required. Furthermore, if binary components or web services are to be discovered there is no source code and thus the search has to use interface descriptions in any case. Considering these differing requirements, a component search engine must be very flexible. However, most of the first generation search engines available today are only able to support keyword-driven searches.

2.1. Component Representation Methods

A repository's component representation format determines the possibilities for searches on this structure. Frakes and Pole [7] identified four basic representation methods briefly explained in the following. *Enumerated classification* originates from library science and separates an area into mutually exclusive, typically hierarchical classes. *Ontologies* in the semantic web community might be considered a modern synonym for this approach. *Facetted classification* [10] and the slightly more general *attribute value classification* approaches are very similar and use a number of facets (resp. attributes) to describe an asset. Each facet comprises a finite set of terms from which one is chosen to describe the asset. In contrast an attribute can contain any arbitrary value. Finally, free text indexing approaches extract textual information from an asset, i.e. the component or its documentation in our context.
There have been a lot of attempts to develop efficient component retrieval techniques for all four approaches. These are best summarized in the well-known survey by Mili et al. [16], but as the authors conclude, *"most*

solutions are either too inaccurate to be useful or too intractable to be usable". Since the representation methods are rather intuitive we turn our attention towards component retrieval techniques that more directly influence the query formulation techniques available to users in the next subsection.

2.2. Component Retrieval Techniques

Mili et al [16] distinguish the following fundamental techniques for component retrieval in five (originally six) not fully orthogonal groups:

1. Information retrieval methods
2. Descriptive methods
3. Operational semantics methods
4. Denotational semantics methods
5. Structural methods

Since software retrieval is grounded on information retrieval, a natural first step was to transfer the methods of the latter to the former and to apply a simple textual analysis to software assets. Descriptive methods go one step further and rely on additional textual descriptions of assets like a set of keywords or facet definitions [10]. Operational semantics methods rely on the execution or so-called sampling [8] of the assets. Denotational semantics methods use signatures [15] or specifications of components for retrieval. Finally, structural methods do not deal with functional properties of components but with their structure (i.e. act-alike vs. look-alike). Most of these mechanisms were initially developed for functional languages with an underlying type theory (i.e. type hierarchies) and no implicit variable passing as is common in today's programming languages. Only a few of these ideas (such as (1) and (2)) are easily transferable to object-oriented languages like Java or C#.
Mili et al. describe a sixth group – the so-called topological methods – as an approach to minimize the "distance" between query and reusable candidates. This approach relies on an underlying, "measurable" retrieval technique and hence we prefer to consider it as an approach for ranking the results of a query. The authors assessed these groups according to a scheme with five discrete rates ranging from very low (VL), low (L) through medium (M) to high (H) and very high (VH). Detailed explanations may be found in the referenced source itself. Due to space constraints, we only reproduce a table containing the three technical aspects that are interesting for the remainder of this paper. *Precision* is a measure from information retrieval (IR) theory that describes the ratio of relevant retrieved assets to the total number retrieved, see e.g. [14]. The *recall* also originates from IR and is the ratio of retrieved relevant assets to the total number of relevant assets in the collection. The meaning of the *automation potential* should be obvious.

Method	Precision	Recall	Automation Potential
1. Information Retrieval	M	H	H
2. Descriptive	H	H	VH
3. Operational Semantics	VH	H	VH
4. Denotational Semantics	VH	H	M
5. Structural	VH	VH	VH

Table 1. Overview of retrieval techniques.

It is important to note that Mili et al. themselves state that due to the low number of publications on this topic their values are largely best effort estimations. Moreover, since there were no large component repositories at that time the data is only based on experience with smaller collections of a few hundred components.

2.2.1. Previous Results

Information Retrieval (IR) typically uses recall and precision as defined previously to evaluate the performance of retrieval systems. Since it is important to know the number of relevant documents in a collection, the IR community has created a number of so-called reference collections (again with about 1000 documents, [20]) over the years. These collections are built by experts and hence it is known which documents can be considered relevant for a given query. Consequently, it is simple to test retrieval algorithms and to calculate recall and precision for them. One of the first authors who investigated the efficiency of their retrieval technique (called "behaviour sampling") in that way were Podgurski and Pierce [8]. They used a small library (around 100 components) of C functions where examples could be retrieved by randomized sampling, i.e. input and output values were provided and functions that delivered the expected outputs for given inputs were considered acceptable. The system delivered a precision of 100 percent if exactly the right number of samples (>= 12) was provided. However, it is clear that this technique is too time consuming for repositories with millions of assets. Frakes and Pole performed an evaluation of retrieval efficiency with UNIX commands [7]. Although it is one of the few publications that focused on this topic it is very domain specific and UNIX commands do not have an interface in the sense of components in modern programming languages. Hence, although these experiments can provide some general insights it is questionable whether they can be generalized and applied to today's retrieval systems. Inoue et al. presented and evaluated a retrieval system [2] that was considerably larger (about 120.000 components) than all previous systems. The authors realized that the classic recall measure cannot be calculated for repositories of that size since it is not possible to find all components potentially relevant for a query. Their examination focused on keyword-based searches (e.g. "clock applet") for Java but unfortunately the authors did not make their relevance criterion explicit. However, they claimed precision rates of about 70 percent for their system. One possible approach to estimate the recall for a large repository is injecting known components into it. However this is not feasible for third party search engines on the web. Moreover, results can easily become biased as they require examples that are known to work well with a given configuration.

To briefly summarize the main issues arising in this context: at present, no reference collections of software components are available. Moreover, even if one were available it could not come anywhere near the size and complexity of today's software repositories. It is therefore questionable whether its results would be scaleable to real world situations. Without the knowledge of how many components are relevant for a query it is not possible to calculate the recall of a search engine. Fortunately, it is feasible to calculate the precision by examining a given number of results and determining whether they do what they are supposed to do. We adopted these insights in our experimental design that we describe in more detail in section 3.

2.3. Component Search Engines

Most of the component search engines available today only offer keyword-based search capabilities based on a free-text representation of components (i.e. they use an IR method). The following table lists the most prominent component search engines that we were aware of at the time of writing.

Name	Languages	Components	Representation Methods Used	Support for Interface-Driven Searches
merobase	45	~10 M	all	yes
Google Codesearch (GCS)	44	~5 M	text, facetted	partially via regular expr.
Krugle	37	~5 M	text, facetted	limited, name-based
Koders	32	~1M	text, facetted	limited, name-based

Table 2. Popular code search engines.

To date, only merobase is able to fully support searches on interface descriptions based on a combination of IR- and denotational methods. Koders and Krugle are able to constrain searches to method or class names (we call this name-based) and interface-driven searches can be widely imitated with regular expressions on Google Codesearch.

The large number of programming languages supported by the engines in table 2 can be explained by the fact that they not only support usual programming languages such as Java and C#, but also index scripting languages like Javascript or PHP and other artifacts such as makefiles etc. We do not consider other search engines such as Codase.com, Planetsourcecode.com, ucodit.com, jsourcery.com, Codehound.com etc. due to their limited size, range of languages or different search focus. As we have shown in [12], it is also possible to use regular Google or Yahoo for source code searches.

3. Experimental Evaluation

Due to the limitations discussed in section 2.2.1 we focused our investigation on the precision of search engines and retrieval techniques. We reused some of our previous work [12] where we collected query examples from the reuse literature. We derived interface-driven queries from them and inspected the first 25 results for Java (as it is most widely supported) from each query in two different experiments. First we evaluated the retrieval performance of various search. The results are presented in section 3.1. Our second experiment performed a more academic comparison of some retrieval techniques and is discussed in section 3.2.

Our matching criterion was that the required interface was contained verbatim or with simply a change of case in a candidate component. In order to finally decide whether such a candidate component is functionally appropriate and thus relevant we improved the sampling approach of [8] and manually defined meaningful JUnit test cases for each interface. We have already experimented with this technique before and found that interfaces and test cases can be used to describe and retrieve components in a very precise manner. Due to its affiliation with test-driven development we have called this approach "Extreme Harvesting". A proof of concept is also presented in [12].

3.1. Comparison of Search Engines

We limited our comparison to the three component search engines shown in table 3 below since only they offered an API for programmatic access at the time of writing. Additionally, we compared them with the general web search versions of Google and Yahoo which we enhanced with special filetype constraints to better utilize them for software component retrieval. To our knowledge, we made the optimal use of each search engines facilities for mimicking interface-driven retrieval. For instance, requiring the term "randomString" to appear only in method names should deliver more precise results with Koders than allowing it anywhere in the source code. Table 3 below summarizes our results for this experiment.

Query	Google	Yahoo	GCS	Koders	merobase
`copyFile(String, String): void`	1 / 25	2 / 25	7 / 25	0 / 25	18 / 25
`gcd(int,int):int`	10 / 25	7 / 25	12 / 25	2 / 25	17 / 25
`isLeapYear(int): boolean`	8 / 25	12 / 25	3 / 25	2 / 25	14 / 25
`md5(String):String`	0 / 25	0 / 25	4 / 22	0 / 25	12 / 25
`isPrime(int): boolean`	6 / 25	15 / 25	7 / 25	4 / 25	5 / 25
`randomNumber(int, int):int`	0 / 25	3 / 25	2 / 7	0 / 7	14 / 25
`randomString(int): String`	4 / 25	2 / 25	6 / 25	4 / 16	5 / 25
`replace(String, String, String): String`	2 / 25	8 / 25	14 / 25	3 / 25	22 / 25
`reverseArray( int[]):int[]`	1 / 10	3 / 23	1 / 1	0 / 4	5 / 7
`sort(int[]):int[]`	0 / 25	0 / 25	5 / 20	0 / 25	20 / 25
`sqrt(double): double`	5 / 25	4 / 25	4 / 25	1 / 25	11 / 25
`getMinMax(int[]): int[]`	0 / 15	0 / 22	0 / 0	0 / 25	2 / 4
`Stack( push(Object):void pop():Object size():int )`	1 / 25	2 / 25	0 / 0	1 / 25	6 / 25
Average Precision	12.2%	17.9%	29.5%	5.9%	53.7%
Standard Deviation	13.3%	18.9%	26.5%	7.8%	22.4%

Table 3. Comparison of code search engines.

We calculated the mean value and the standard deviation of each engine's precision. Furthermore, we performed t-tests for $\alpha = 0.05$ to measure the statistical difference of the results. Only the results provided by merobase show a significant improvement over the other engines. Google Codesearch (GCS) is also significantly better than Koders; but all other pairwise comparisons reveal no statistically significant difference. It is interesting that the general versions of Google and Yahoo tend to deliver more precise results for code searches than the specialized engine of Koders. However, we believe this can be explained by the different expressiveness of the queries offered by the different search engines. We will support this with more evidence in the next subsection where we directly compare retrieval methods.

3.2. Comparison of Retrieval Techniques

This subsection presents the results of our experiments to compare four retrieval techniques on the component pool of merobase. The experimental process is identical to that used in the last subsection. Since we had access to the data pool of merobase and could implement some dedicated support for these experiments we used this engine for a comparison of the retrieval techniques shown in table 4. However, it would not have been possible to test other known retrieval techniques or the representation models on their own without major changes to index structure. Hence we compared interface-driven search capabilities with pure signature matching and simple keyword-based searches in two distinct forms. Namely, an algorithm that searches keywords in the complete source code of components and a name-based algorithm that is able to constrain searches to method or class names (cf. table 2).

Query	signature matching	text-based	name-based	interface-driven
`copyFile(String, String): void`	0 / 25	3 / 25	16 / 25	18 / 25
`gcd(int,int):int`	0 / 25	20 / 25	11 / 25	17 / 25
`isLeapYear(int): boolean`	0 / 25	9 / 25	7 / 25	14 / 25
`md5(String):String`	0 / 25	0 / 25	0 / 25	12 / 25
`isPrime(int): boolean`	0 / 25	4 / 25	5 / 25	5 / 25
`randomNumber(int, int):int`	0 / 25	0 / 25	0 / 25	14 / 25
`randomString(int): String`	0 / 25	4 / 25	6 / 25	5 / 25
`replace(String, String, String): String`	1 / 25	6 / 25	0 / 25	22 / 25
`reverseArray( int[]):int[]`	0 / 25	0 / 25	2 / 25	5 / 7
`sort(int[]):int[]`	1 / 25	0 / 25	0 / 25	20 / 25
`sqrt(double): double`	0 / 25	2 / 25	4 / 25	11 / 25
`getMinMax(int[]): int[]`	1 / 25	2 / 25	2 / 25	2 / 4
`Stack( push(Object):void pop():Object size():int )`	0 / 25	3 / 25	3 / 25	6 / 25
Average Precision	0.9%	16.3%	17.2%	53.7%
Standard Deviation	1.8%	21.9%	19.3%	22.4 %

Table 4. Comparison of retrieval techniques.

We again performed statistical t-tests for $\alpha = 0.05$ on these results and found all pairwise comparisons significantly different, except for text-based vs. name-based. The results demonstrate that interface-driven searches are far more precise than plain keyword-based queries. This might also explain why Koders tends to be weaker then the general versions of Google and Yahoo where interface-driven searches can be better "simulated" with quoted queries. Google Codesearch and merobase consequently deliver significantly better results when their capabilities for regular expressions or interface-driven searches are used. However the precision remains still roughly between 30 and 60 percent and given the fact that sometimes thousands of results are returned a further increase of the precision is still required. This can be obtained by the use of a final semantic validation as integrated in our Extreme Harvesting approach where we use standard JUnit tests to check the dynamic behavior of components.

Another important requirement for precise searches are so-called search constraints that allow queries to be enhanced with additional metadata (such as the programming language) as is common in most general web search engines today. Thus, we believe it is not only necessary to combine various of the retrieval techniques proposed in the literature to reach acceptable precision on today's component collections, but also to combine a number of representation methods.

4. Conclusion

In the last year or so there has been an explosion in the number of search engines focusing on the discovery of software. However, their search algorithms and degree of precision are generally too weak to deliver a valuable service. One retrieval technique alone is typically not sufficient to guarantee high precision searches on a large repository and hence it makes sense to develop a combination of various techniques as we proposed for Extreme Harvesting [12]. However, this idea has so far been largely based on our intuition since we had no adequate repository at hand to compare the retrieval techniques proposed in the past. The results presented in this paper demonstrate that interface-driven searches deliver significantly better results than simple keyword-based approaches. Furthermore they deliver better candidates that can be checked with a more time consuming retrieval technique such as behavior sampling or the more advanced Extreme Harvesting. Testing of the form advocated in Extreme Harvesting seems to be able to push the precision close to 100%. However, to make Extreme Harvesting practicable we still have to overcome a number of practical problems (such as security and performance concerns). We are currently working on this challenge and will elaborate on further experiments on another occasion.

5. References

[1] C.W. Krueger, "Software reuse", *ACM Computing Surveys*, Vol. 24, Iss. 2, 1992.

[2] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, S. Kusumoto, "Ranking Significance of Software Components Based on Use Relations", *IEEE Trans. on Software Eng*., Vol. 31, No. 3, 2005.

[3] Clemens, P., Northrop, L.: *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] D. McIlroy, "Mass-Produced Software Components", *Report of a conference sponsored by the NATO Science Committee*, Garmisch, 1968.

[6] Microsoft's UBR shutdown FAQ, 2006, http://uddi.microsoft.com/about/FAQshutdown.htm

[7] W.B. Frakes, T.P. Pole, "An empirical study of representation methods for reusable software components", *IEEE Transactions on Software Engineering,* Vol. 20, Iss. 8, 1994.

[8] Podgurski, A., L. Pierce: "Retrieving Reusable Software by Sampling Behavior", *ACM Transactions on Software Engineering and Methodology*, Vol. 2, Iss. 3, 1993.

[9] M. Lenz, H. Schmid, P. Wolf, "Software reuse through building blocks", in W. Tracz (ed..): *Software Reuse: Emerging Technology*, Computer Society Press, 1987.

[10] R. Prieto-Diaz, "Implementing Faceted Classification for Software Reuse". *Communications of the ACM*, Vol. 34, Iss. 5, 1991.

[11] I. Crnkovic, M. Chaudron, S. Larsson, "Component-based Development Process and Component Lifecycle", *Proceedings of Int. Conf. on Software Engineering Advances*, 2006.

[12] O. Hummel, C. Atkinson, "Using the Web as a Reuse Repository", *Proceedings of the International Conference on Software Reuse*, Torino 2006.

[13] R. Seacord: "Software Engineering Component Repositories", *Proceedings of the Int. Workshop on Component-Based Software Engineering*, 1999.

[14] R. Baeza-Yates, B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[15] A.M. Zaremski, J.M. Wing: "Signature Matching: A Tool for Using Software Libraries", *ACM Transactions on Software Engineering and Methodology*, Vol. 4, No. 2, 1995.

[16] Mili, A., R. Mili and R. Mittermeir: "A Survey of Software Reuse Libraries", *Annals of Software Engineering,* Vol. 5, 1998.

[17] K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

[18] Y. Ye., G. Fischer, "Reuse-Conducive Development Environments", *Journal of Automated Software Engineering*, Vol. 12, No. 2, 2005.

[19] J. Poulin, "Populating Software Repositories: Incentives and Domain-Specific Software", *Journal of Systems and Software,* Vol. 30, Iss. 3, 1995.

[20] Y.S. Maarek, D.M. Berry, G.E. Kaiser, "An information retrieval approach for automatically constructing software libraries", *IEEE Trans. on Software Engineering*, Vol. 17, Iss 8, 1991.

[21] W.J. Brown, R.C. Malveau, H.W. McCormick, T.J. Mowbray, AntiPatterns: refactoring software, architectures, and projects in crisis, Wiley, 1998.

[22] B. Kratz, R. Reussner, W.J. v.d. Heuvel, "Empirical Research on Similaritiy Mertrics for Software Component Interfaces", *Journal of Integrated Design and Process Science*, Vol. 8, Iss. 4, 2004.