Using the Web as a Reuse Repository

Oliver Hummel and Colin Atkinson

University of Mannheim, Chair of Software Technology
68159 Mannheim, Germany
{hummel, atkinson}@informatik.uni-mannheim.de
http://swt.informatik.uni-mannheim.de

Abstract. Software reuse is widely recognized as an effective way of increasing the quality of software systems whilst lowering the effort and time involved in their development. Although most of the basic techniques for software retrieval have been around for a while, third party reuse is still largely a "hit and miss" affair and the promise of large case component marketplaces has so far failed to materialize. One of the key obstacles to systematic reuse has traditionally been the set up and maintenance of up-to-date software repositories. However, the rise of the World Wide Web as a general information repository holds the potential to solve this problem and give rise to a truly ubiquitous library of (open source) software components. This paper surveys reuse repositories on the Web and estimates the amount of software currently available in them. We also briefly discuss how this software can be harvested by means of general purpose web search engines and demonstrate the effectiveness of our implementation of this approach by applying it to reuse examples presented in earlier literature.

1 Introduction

It has long been recognized that reuse is the key to making software development a fully fledged engineering discipline [19] in which quality systems are built at low cost in a dependable and predictable manner. In principle, almost all assets that are produced during a software development process such as domain knowledge, requirements, design and source code are potentially reusable. Traditionally, however, reuse initiatives have focused on the reuse of software in binary or source-code form [1]. Even reuse in this sense is an umbrella term for many different concepts that can range from ad-hoc copying of a few lines of code to the architecture-centric usage of large parts of a software product line [12]. In this paper our focus is on software components, but not in the sense of Szyperski [13] which emphasizes binary software units, rather as source code units that can be used independently. This can range, in the simplest case, from a class that contains a stateless method, to a variety of classes that depend on some shared libraries. Unfortunately, these more complex forms of components are difficult to retrieve since common programming language do not make their required interfaces explicit.

There has been an ongoing discussion in the literature (see e.g. [3], [6]) over whether a component repository is a necessary condition for a successful reuse program or not. Failure mode analyses have established that to be reused a component must at the very minimum be available and findable, and component repositories are certainly one way of achieving this [24]. It has often been argued that typical reuse collections are small and hence do not need library support, however, intuition suggests that the bigger the component collection the higher the probability it contains a matching artifact [30]. Given this observation it makes sense to study the ability of repository organization and retrieval techniques to handle large component collections. To date Mili et. al.'s well known survey [17] gives the best overview of this topic. After their study Mili et. al, like Seacord [6], were rather pessimistic that there will be a solution to the so-called "repository problem" in the foreseeable future. They argue that currently "(...) *no solution offers the right combination of efficiency, accuracy, user-friendliness and generality to afford us a breakthrough in the practice of software reuse"*.

In addition to these academic studies of software reuse, there have been numerous attempts to establish commercial component "marketplaces" in recent years. However, these have also had limited success. Two of the most well known, ComponentSource.com and Flashline.com, have had to merge recently. Moreover, the Universal UDDI Business Registry (UBR), the high profile industry repository for web services, rarely contained useful material (as we will show later) and was finally shut down in January 2006[1]. Likewise, most other initiatives have had very limited impact. These stated approaches have essentially all been based on a standard "e-retail" model in which components are offered in an informal catalogue-like style as if they were mainstream consumer products. Trying to discover a component at ComponentSource is therefore still much like browsing for a book on Amazon. It is a very informal, unpredictable process with a highly uncertain outcome. Of course, searching tools are provided, but these are very simple, typically text-based technologies which essentially look for keywords in a component's documentation.

1.1 The Opportunity

Naturally the rise of the Internet as a public library for almost everything has raised the reuse community's interest in utilizing it for their purposes (see e.g. [17], [6] & [9]). In recent years there have been a growing number of research projects that have made initial steps towards this goal. The earliest known approach that utilized the Web together with a general purpose search engine was Agora [6]. Other researchers and commercial websites have crawled publicly available CVS repositories to build their own source code search engines (SPARS-J [25], Koders.com, Codase.com) or for other research purposes (for instance [15]). Others have recently experimented with the use of general search engines (such as Google and Yahoo) to search for components. However, [25] did this only in a rudimentary way by augmenting queries with the terms "java" and "source" while [29] questioned the feasibility of doing this. Despite this pessimism, we have succeeded in developing a reuse approach called Extreme Harvesting that we first introduced in [2] that can successfully retrieve components from the Web. The basic idea is to use the whole Web itself as the underlying repository, and to utilize standard search engines as the means of discovering appropriate software assets. Since the Web, by its very nature, is a very unstructured and

[1] The official rationale is that the UBR has been successful as a proof of concept, though.

unruly place that was not designed to store software source code this is not always easy. However, we have shown it is indeed possible to automatically harvest all kinds of valuable components by means of general search engines.

As the main purpose of this paper is to assess the size and quality of the Web as a software repository we give only a brief overview of our Extreme Harvesting approach in the next section. Section 3 surveys specialized service and software search engines on the web and evaluates their efficiency. In section 4 we compare these results with the outcome of our Extreme Harvesting experiments and give our assessment of the Web's potential to serve as a ubiquitous software repository. Finally, in section 5 we conclude and discuss potential future directions of the work.

2 Component Retrieval Basics

For the reader to understand why it makes sense to search the Web for usable software components despite the problems described in [29] and above we briefly introduce our Extreme Harvesting approach. Based on the lessons learned from Mili et al.'s survey [17] and our own experiments we created this new hybrid semantics-driven retrieval engine by integrating some of the techniques outlined there. As stated in the survey, a retrieval process typically has to cover two criteria because a candidate component can fulfill the matching condition of one specific retrieval technique but may not necessarily match a user's relevance criterion. Consider the above mentioned keyword-based search technique, for instance. Such a search engine might retrieve a number of components that contain the word Stack somewhere (maybe they use a Stack), but only very few of them implement the appropriate data structure. In other words, a single matching criterion is too weak to guarantee satisfactory precision.

Applying more than one matching criterion essentially represents a filtering process that iteratively shrinks the number of acceptable components in a repository search until only acceptable components are left. In our current tool we apply three filtering stages, namely linguistic, syntactic and semantic filtering. The linguistic filtering is basically a keyword search as described above. After that a signature matching step is applied [22]. Then, thirdly, we check the semantic compliance of components by sampling their behavior [7]. As we have focused our current research on Java we chose quasi standard JUnit [23] test cases to represent this information. Unfortunately, behavior sampling of this from is only a limited substitute for complete semantic checking, but it is the only practical way at present, because to find out whether a code unit complies to a given formal description is equivalent to solving the halting problem [28].

The cost of applying these filtering steps grows in the order they are introduced. For this reason the combination of the three steps is the only practical way to retrieve components with reasonable precision from very large repositories like the web. In other words, it would never be computationally possible to apply a semantic relevance check to millions of components. Figure 1 below provides a schematic summary of the main steps involved in the practical implementation of our approach as originally introduced in [2]:

a) define syntactic signature of desired component
b) define semantics of desired component in terms of test cases
c) search for candidate components using the APIs of standard web search engines with a search term derived from (a)
d) find source units which have the exact signature defined in (a) or try to create appropriate adapters
e) filter out components which are not valid (i.e. not compilable) source units, if necessary find any other units upon which the matching component relies for execution
f) establish which components are semantically acceptable (or closest to the requirements) by applying the tests defined in (b)

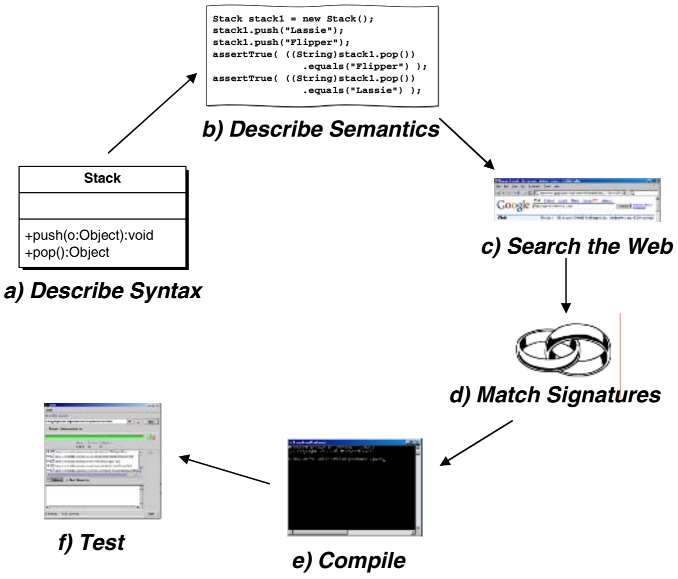

```
Stack stack1 = new Stack();
stack1.push("Lassie");
stack1.push("Flipper");
assertTrue( ((String)stack1.pop())
               .equals("Flipper") );
assertTrue( ((String)stack1.pop())
               .equals("Lassie") );
```

Fig. 1. Schematic process overview

We currently have a Java-based prototype which implements the above approach and is able to harvest Java components and web services from the web. Extending the tool to handle other programming languages is a straightforward matter. Since our three step filtering process has proven to be very effective in our experiments and our tool has the capability to adapt search results into the form required in (a) automatically, a developer can integrate any accepted search result right away in his/her development project.

3 Internet-Based Repositories

This section briefly reviews the component and service search engines available on the Internet or reported in the literature. Since one of the reasons for the recent excitement around web service technology is that its search technology UDDI [16] is supposed to bring together service providers and service requestors we start our overview with web services that are available for third-party (re-)use. UDDI is advertised as a flexible brokering technology that allows component developers to "publish" their software as services, and potential component users to find suitable services automatically through formalized syntactic descriptions of their requirements (in the form of WSDL documents). Even semantic composition capabilities for web services are becoming available (e.g. with the help of OWL [8]). Since so much industry investment has been pumped into the Universal UDDI Business Registry (UBR) one would expect a sizeable index of services to now be available. However, as table 1 demonstrates, the UBR (and other service repositories) failed to reach a critical mass of entries and a large proportion of the entries contained in the repository were out of date. Many entries did not even point to valid WSDL descriptions and of those that did, only a small proportion were actually working. The UBR's shutdown in early 2006 was a logical consequence.

Table 1. Number of WSDL files within reach at various websites (July 2005)

Search Method	API	Claimed number of links to WSDL files	No of actual links to valid WSDL files
UDDI Business Registry[2]	yes	770 (790 [11])	400 (430 [11])
BindingPoint.com	no	3900	1270 (validated)
Webservicelist.com	no	250	unknown
XMethods.com	yes	440	unknown
Salcentral.com[2, 3]	yes	~800	all (validated)

We can only speculate about the reasons for the disappointing performance of such repositories. However, the main problem with this concept in our opinion is not a technical one, it is the overhead involved in the manual creation and maintenance of the repository. The effort involved in entering a complete service profile into the UBR should not be underestimated. In addition, there is the effort of updating or removing the (possible many) entries when a server is moved or closed down. Interestingly, the UBR followed exactly the three-phase reuse progression (empty, filled with garbage or too general) that Poulin reported in [30] from his practical experience at IBM over ten years ago (although we would argue that the UBR actually never reached the third phase).

Since the Web in its current form is still relatively new there have been few attempts to date to utilize it as a source of components for mainstream software engineering. The most well known attempt is Agora search engine [6] mentioned above. Agora was developed at the Software Engineering Institute (SEI) as a special purpose search engine with its own index of Java applets and ActiveX components which has

[2] As of March 2006 this website is no longer available.

[3] Salcentral copied the entries of UDDI and XMethods, the values were estimated from active UDDI and XMethods entries.

been filled using a general purpose search engine. However, this project was discontinued probably due to the high effort involved in setting up the index. In addition to this approach, focused on black-box components, the recent advent of open source software has also made it possible to look – at least manually – for white-box components – that is, publicly available source code on the Web [26].

Another idea, utilized by a number of papers in the 2004 and 2005 ICSE workshops on mining software repositories, is to crawl the CVS servers of Sourceforge.net or similar sites (see e.g. [15]) and analyze the content in some way. However we are not aware of an approach that explicitly aims to reuse this material. As Sourceforge does not offer search capabilities for the code it stores, the approach of Koders.com, a fairly new commercial site, makes a lot of sense. They download and index source codes from publicly available CVS repositories and then support text based searches on these assets through a Google style search interface. Codase.com has built a similar index that offers limited support for syntactic searches constrained to method names or parameter types. Krugle.com is a similar site that is scheduled to come on line early in 2006. The following table provides an overview of the sites known to us at the time of writing. We did not consider software retailers like Component-Source.com or Jars.com in this overview as they typically offer very large packages or complete applications which are beyond the scope of our approach and do not offer access to source code.

Table 2. Overview of specialized source code search engines (January 2006)

URL	**No. of Languages supported**	**API**	**Supported Search Methods**	**Indexed Lines of Code**	**No. of Java classes**
Koders.com	30	RSS	Linguistic	225,816,744	330,000
demo.spars.info [25]	1	no	Linguistic & Syntactic	n.a.	180,000
Kickjava.com	1	no	Linguistic	20,041,731	105,000
Codase.com	3	no	Linguistic & Syntactic	250,000,000	95,000
Csourcesearch.net	2	no	Linguistic & Syntactic	283,119,081	n.a.
Sourcebank.com	8	no	Linguistic	n.a.	> 500
Planetsourcecode.com	11	no	Linguistic	11,276,847	230

In contrast to general web search engines the listed sites are specialized for source code searches. Hence, they all offer the opportunity to limit searches to a specific language, but only Koders.com fulfills another important requirement for being accessible with our tool, namely an API for programmatic access. Their API is based on Amazon's Opensearch format which in turn is based on RSS. As illustrated by the table above, none of the listed sites provides a form of semantic evaluation for the searches and only a few support the constraining of queries to given syntactic elements (such as method names or parameter types). The estimates we provide for the size of the repositories are the number of indexed lines of code (where this is specified on the site) and the number of Java classes available (by searching for the term "class" in Java files).

4 The Web as a Component Repository

In section 2 we described how a suitable combination of well known techniques and heuristics can effectively harvest components from the web when the desired kind of component is present. However, as with any component repository, it cannot deliver components if there are no suitable ones in the repository. As discussed in section 3, this has been highlighted by web services, the most recent attempt to make third party software components discoverable and accessible via the Internet. As shown in table 1 the Universal UDDI Business Repository has fallen far short of the original predications. Other specialized source code search engines are better, but still only deliver a small part of the Web's potential as we will demonstrate below.

The effectiveness of the retrieval mechanism is only one prerequisite for a practically useful reuse technology. The other is the availability of a repository with a rich and extensive collection of components which covers a large proportion of the kinds of components that users are likely to require [6]. In this section we discuss and evaluate the extent to which the web is able to fulfill this need. As briefly mentioned above, search engines are appropriate for integration in an automated approach like ours if two prerequisites are satisfied. First, a search engine must have an API that allows computational access to its index and second – and this is very important for general search engines as Google and Yahoo – there must be a way to (pre-) filter searches according to a given programming language. To date we have found these features in three engines, namely the two market leaders for general web searches Google and Yahoo where we are able to exploit an undocumented feature of their "filetype" filter, and the specialized engine from Koders.

4.1 Repository Volume

To illustrate the magnitude of the accessible code resources on the web the following table shows the numbers of Java files that could be retrieved using Google, Koders and Yahoo search engines during our experiments in 2004 and 2005. Two sets of values are shown for the Google entries – the first giving the number obtained using the regular human HTML interface and the second (bracketed) giving the number obtained using the Web-API for automated access. Unfortunately, the latter delivers only a fifth of the results available using the former.

The italicized value in the last row stems from the query *"filetype:java" class OR –class*. One should assume that a search with *"filetype:java" -class* only delivers Java interfaces and no classes but actually this is not the case. Manual inspections revealed a high percentage of class files. One explanation for this strange result may be that Google does not completely index some files. The numbers in the table represent the mean value of several samples per month whereas individual values can vary even

Table 3. Number of Java files indexed by search engines on the Web

Month	Google (Web API)	Koders	Yahoo
08/2004	300,000	-	-
01/2005	640,000	-	-
06/2005	950,000 (220,000)	310,000	280,000
08/2005	970,000 (220,000) *1,510,000 (367,000)*	330,000	2,200,000
11/2005	2,212,000 (190,000) *4,540,000 (410,000)*	330,000	2,200,000

from one request to the next within just a few minutes (for Google and Yahoo). However, the growth trend illustrated by the numbers is unmistakable. In August 2005 a similar request for various C-style languages (filetypes: c, cpp and cs) revealed a total of about 1.6 million source files in Google's index, 2.7 million from Yahoo and 500,000 from Koders.

The overlap between Google and Yahoo seems to be rather low - it is typically below 20% (5 out of 24) for our `isLeapYear` example (see table 6) and for the first 250 results of each engine for our `Matrix` example from table 7, 47 out of 500 overlap. This observation tallies with other reports for general HTML searches [14]. Unfortunately, it is not possible to estimate a URL-based overlap between Koders and Google/Yahoo because Koders stores the contents with proprietary URLs. With the numbers presented above, we estimate that our system currently has access to about 3 million Java files. This is – to our knowledge – the most comprehensive source-code collection reported in the literature so far. Inoue et. al. [25] has access to roughly 180,000 classes and Agora to around 10,000 (black-box) applets [6].

Similar to Agora, Yahoo allows a search to be limited to pages that contain Java applets (*feature:applet*), delivering the impressive number of 95,000,000 results, or to ActiveX components (*feature:activex*), resulting in an astonishing 750,000,000 pages. Although our tool focuses on white-box components at present, it should be possible to use mechanisms like Java's reflection capability to utilize this large number of black-box components as well. Initial experiments in this direction have already demonstrated promising results: we were able to populate a database with more than 4500 JAR files containing almost 500,000 classes.

Google and Yahoo could also be helpful for the web service community since they are also able to retrieve WSDL files. As the next table illustrates, they are actually better at discovering WSDL files than the web service repositories from table 1.

Table 4. Number of WSDL files delivered from search engines

Search Engine	API	Claimed no. of links to WSDL files	No. of actual links to valid WSDL files
Google	yes	9000 (1700)	794 out of first 1000
Yahoo	yes	13400 (1900)	425 out of first 1000

The values in brackets show the number of results returned through the APIs. This indicates that the search results could be better were not it for the artificial limitation imposed on automated queries. Both search engine APIs allow access to only the first 1000 results returned in response to a query. This is not usually a problem when searching for a specific functional component since the number of retrieved candidates rarely exceeds a few hundred. To conclude this subsection, we summarize the results of our investigations in the following table. This reinforces our belief that the Web has a high, but so far neglected potential as a software repository.

Table 5. Summary of investigated component types that are accessible via an API

Type	Estimated number	Applicable search engines
.java	3,000,000	Google, Yahoo, Koders
.c, .cpp, .cs	4,000,000	Google, Yahoo, Koders
.wsdl	10,000	Google, Yahoo, UDDI, BindingPoint, XMethods
.jar	600,000	Yahoo
Applets	95,000,000	Yahoo
ActiveX	750,000,000	Yahoo

4.2 Repository Scope

Beyond the shear number of components the functional scope of the components in a repository is another interesting characteristic which is a widely unexplored issue in the reuse literature. Most reuse approaches published to date provide some kind of estimate of their tool's power. Typically, however, the underlying repositories used in such evaluations only contained up to a few thousand classes with very limited scope. Furthermore, their comparability is very low since most evaluations were based on proprietary repositories supporting some special features tailored to the employed retrieval technique. Moreover, in order to get any results from these experiments researchers had to give tasks to their subjects that were indeed solvable with the repositories contents. As one possible solution for this issue we propose the definition of reference collections of the kind commonly used in information retrieval research to evaluate "standard" retrieval systems. However, due to the high complexity of, and large variations in, software solutions it is clear that this will not be easy.

Another issue arises with the assessment of uncontrolled repositories like the Web. It is very likely – as confirmed by our experiments – that large numbers of components with common functionality appear on the Web. This is of course ideal for reuse. However, it compounds the problems involved in comparing retrieval techniques and estimating the scope of a software repository. Our solution for this problem was to take examples from comparable reuse experiments to (a) get an impression of the quality of our combination of retrieval techniques and (b) to estimate the scope of the Web as a repository. Another insight into the demand for component searches was provided by the Koders' search statistics[4]. The table below gives an impression of the capability of our tool and shows that it compares favorably to other approaches. The table presents various stateless components that offer typically used algorithms. The first column presents the method names that we used for the search, the second column shows the signature that we entered into our system. Columns three, four and five show how many results passed the filtering process and the last column shows the source which provided the inspiration for the example. Due to space restrictions we cannot show the test cases for the semantic checking here. It should be enough to know that we used about three to five test cases per example as they are typically applied for unit testing in non-reuse processes.

Table 6. Query results from June and July 2005

Names	**Signature**	**Koders**	**Yahoo**	**Google**	**Source**
getRandomNumber	int x int: int	3	6	2	[5], [25]
sort	int[]: void	1	12	15	Koders
reverseArray	int[]: void	0	10	6	-
copyFile	String: void	2	1	0	Koders
isPrime	int: Boolean	1	8	14	[18]
sqrt	double: double	2	9	5	[7]
isLeapYear	int: Boolean	1	29	24	[5]
replace	String x String: String	14	10	22	Koders
gcd[5]	int x int: int	3	68	10	[10]
md5	String: String	3	1	0	Koders
lcs[6]	String x String: String	0	0	2	[10]
quicksort	String[]: void	4	3	2	[25]

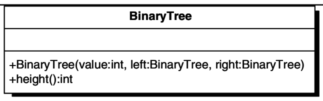
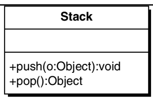
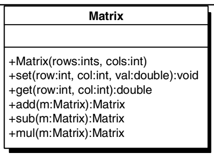
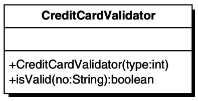
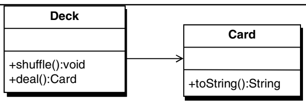
Due to the heuristics implemented in our prototype, results with slightly different names were adapted to the original signature and also accepted, like `getRandomInt` instead of `getRandomNumber` and so on. Furthermore, the autoboxing capabilities of Java 1.5 came handy for the `BinaryTree` example from the table below which illustrates more complex and typically stateful components. Interestingly, we were not able to retrieve a single functioning web service for any of the examples from table 6 above, and we were only able to find the `CreditCardValidator` from table 7 with more complex classes below. We describe the interfaces of these examples in the form of UML class diagrams:

[4] http://koders.com/info.aspx?page=LanguageReport

[5] Greatest common divisor.

[6] Longest common substring.

Table 7. Exemplary stateful components

Component's UML diagram	Koders results	Yahoo results	Google results	Source
 BinaryTree +BinaryTree(value:int, left:BinaryTree, right:BinaryTree) +height():int	0	4	7	[17]
	6	13	33	[25] & similar to [22]
 Matrix +Matrix(rows:ints, cols:int) +set(row:int, col:int, val:double):void +get(row:int, col:int):double +add(m:Matrix):Matrix +sub(m:Matrix):Matrix +mul(m:Matrix):Matrix	1	1	3	[21]
 CreditCardValidator +CreditCardValidator(type:int) +isValid(no:String):boolean	1	1	1	[20]
 Deck +shuffle():void +deal():Card Card +toString():String	-	20	17	[5]

4.3 Component Quality

The most pressing question still to be answered is of course the quality of components downloaded from the Web. So far we found that most components that passed our tests were of reasonable quality, and some minor problems (e.g. with the isLeapYear example or the size of harvested Stack classes) could have been avoided with better test cases. This directly leads to the realm of reliability measurement and the evaluation of components to certain levels of confidence. Even when a component passes all tests defined by a developer it is not certain that it will perform with 100% reliability since unit tests are incomplete in most practical situations. As this is also the case for non-reuse components, further acceptance tests would certainly follow in either case.

However, as harvesting typically delivers multiple results for a search request the idea of back-to-back testing [27] (i.e., comparing the results of functionally identical components for the same random input) is a good starting point to estimate the reliability of retrieved components. This naturally leads to another area of enhancement which relates to the issue of ranking components. At present the result of our selection process is a list of components which have passed all the filtering steps and thus qualify as "working" components. However, this set is not ordered in any way. The next logical extension of the approach is to present the components in a ranked list similar to that of Google and Spars-J [25]. There are many possible ways of doing this like depending on non-functional attributes of a component such as its estimated reliability or code metrics to mention just a few.

4.4 Extensibility

One way of estimating the size of the World Wide Web as a component repository is to inject known components into and determine how easily they can be detected. One way of doing this is to insert files into the CVS repository of a big open source site like Sourceforge since these are almost immediately made available on the Web. Another approach would be to simply store source files on a web server, link them via a HTML file and submit everything to the crawlers of one of the big search engines. We did exactly this in early 2006 with some Java projects. However, the results were not encouraging. Google had not indexed any of them in our eight week observation period and via Yahoo our index page was accessible for a few days but was then removed again. A possible explanation might be that the big search engines focus on human readable material and hence try to avoid including source code in their index. Koders also appears not to have updated its index for many months. These observations make it clear that contributing to the ubiquitous repository World Wide Web in a controlled fashion is not practical at present.

We have also investigated whether the common peer-to-peer (P2P) platform Gnutella is useful for component distribution, as P2P systems are typically a place where all kinds of files can be easily shared with almost no effort. However, the results are – at least currently – not encouraging. For instance, there are only about 2,500 Java source files available in the Gnutella network on average. And as P2P systems simply search in the name and not in the content of files they offer only the most simplistic search support and hence offer not much incentive for developers to use P2P systems for this purpose. These investigations show that there is plenty of room for a dedicated P2P or web search system that makes it easy contribute code, perhaps in the same way that CVS plug-ins for common CASE tools function.

5 Conclusion and Future Work

There have been many notable attempts during the history of computer science to make software reuse a more integral part of industrial software engineering, but to date they have all foundered on the problem of creating and maintaining a sufficiently rich and large repository of components. This includes the UDDI-based Universal Business Registry which despite the relative newness of the technology was full of unusable material before it has been closed down recently.

In contrast to this experience, the contribution of this paper is to show that (1) the Web has become sufficiently large and stable to serve as a self-maintaining component repository and (2) that it is possible to build an engine which can harvest components from this repository in an efficient and dependable way. Since we are still in a fundamental research stage there is a whole host of other issues to be addressed. The security problem associated with executing unknown software from the Web is one example, of course. Hence, we are working to extend the capabilities of our prototype tool in this and several other directions. Support for some kind of ontology or thesaurus technology is one important idea. Another is the inclusion of proactive recommendation technology in the spirit of CodeBroker [25]. Although our approach originated from agile development approaches we also aim to provide tight integration into modern component development methodologies like KobrA [12]. Closely related to this aspect is the problem that common programming languages do not make components they rely on explicit- that is, their required interface is typically hidden inside the source code. Although, we have made good progress in resolving the required interfaces of components (i.e. the imports of Java files) there is still a long way to go. Finally, there are lots of ethical and legal aspects related to the harvesting of software from the Web that could also influence the usability of a component. However, as with most other Internet technologies including search engines and peer-to-peer file sharing systems, the technology usually comes first and the legal issues are sorted out afterwards. Therefore, we hope the work described in this paper will provide a new impulse to software reuse and will help bring closer the day when automated access to a rich library of software components is the rule rather than the exception.

References

1. McIlroy, D.: Mass-Produced Software Components. Software Engineering: Report of a Conference sponsored by the NATO Science Committee, Garmisch (1969)
2. Hummel, O., Atkinson, C.: Extreme Harvesting: Test Driven Discovery and Reuse of Software Components, Proceedings of the International Conference on Information Reuse and Integration (IEEE-IRI), Las Vegas (2004)
3. Frakes, W. B., Fox, C.J.: Sixteen Questions about Software Reuse. Communications of the ACM, Vol 38 Issue 6 (1995)
4. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (1999)
5. Ye, Y., Fischer, G., Reuse-Conducive Environments. Journal of Automated Software Engineering, Vol. 12, Iss. 2, Kluwer (2005)
6. Seacord, R.: Software Engineering Component Repositories, Proceedings of the International Conference of Software Engineering, Los Angeles (1999)
7. Podgurski, A., Pierce, L.: Retrieving Reusable Software by Sampling Behavior. ACM Transactions on Software Engineering and Methodology, Vol. 2, Iss. 3 (1993)
8. Sirin, E., Hendler, J., Parsia, B.: Semi-automatic composition of web services using semantic descriptions. In Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2003, Angers (2003)
9. Frakes, W.B., Kang, K.: Software Reuse Research: Status and Future. IEEE Transactions on Software Eng., Vol. 31, No. 7 (2005)
10. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, 2nd Edition. MIT Press (2001)
11. Dong, X., Halevy, A., Madhavan, J., Nemes, E., Zhang, J.: Similarity Search for Web Services. Proceedings of the 30th VLDB Conference, Toronto (2004)
12. Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., Zettel, J.: Component-based Product Line Engineering with UML. Addison Wesley (2002)
13. Szyperski, C.: Component Software, Addison-Wesley, 2nd Edition, 2002
14. Dogpile.com: Different Engines, Different Results, Technical Report, (2005): http:// comparesearchengines. dogpile.com/ OverlapAnalysis.pdf (accessed 09/08/05).

15. Amin, R., Ó Cinnéide, M. and Veale, T.: LASER: A Lexical Approach to Analogy in Software Reuse, Proceedings of the International Workshop on Mining Software Repositories, Edinburgh (2004)
16. Belwood, T., Clément, L., Ehnebuske, D., Hately, A., Hondo, M., Husband, Y., Januszewski, K., Lee, S., McKee, B., Munter, J., von Riegen, C.: UDDI Version 3.0. Oasis Committee Specification (2002)
17. Mili, A., Mili, R., Mittermeir, R: A Survey of Software Reuse Libraries. Annals of Software Engineering 5 (1998)
18. Hall, R.J.: Generalized behavior-based retrieval. Proceedings of the International Conference on Software Engineering, Baltimore (1993)
19. Mili, A., Yacoub, S., Addy, E., Mili, H., Toward an engineering discipline of software reuse. IEEE Software, Vol. 16, No. 5 (1999)
20. Vitharana, P., Zahedi, F., Jain, F.: Knowledge-Based Repository Scheme for Storing and Retrieving Business Components. IEEE Transactions on Software Engineering, Vol. 29, No. 7 (2003)
21. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison Wesley (2000)
22. Zaremski, A.M. Wing, J.M.: Signature Matching: A Tool for Using Software Libraries. ACM Transact. on Software Engineering and Methodology, Vol. 4, No. 2 (1995)
23. Beck, K., Gamma, E., JUnit: A Cook's Tour. Java Report (August 1999)
24. Frakes, W.B., Fox, C.J.: Quality Improvement Using A Software Reuse Failure Modes Model. IEEE Transactions on Software Eng., Vol. 22, No. 4 (1996)
25. Inoue, K., Yokomori, R., Fujiwara, H., Yamamoto, T., Matsushita, M., Kusumoto, S.: Ranking Significance of Software Components Based on Use Relations. IEEE Transactions on Software Eng., Vol. 31, No. 3 (2005)
26. Brown, A.W., Booch, G.: Reusing Open-Source Software and Practices: The Impact of Open-Source Software on Commercial Vendors. In C. Gacek (Ed.): LNCS 2319, Springer (2002)
27. Vouk, M.A., Back-to-Back Testing. Information & Software Techn., Vol. 32, No. 1 (1990)
28. Edmonds, B., Bryson, J.: The Insufficiency of Formal Design Methods - the necessity of an experimental approach for the understanding and control of complex MAS, Proc. of the 3rd Intern. Joint Conf. on Autonomous Agents & Multi Agent Systems, New York (2004)
29. Yao, H., Etzkorn, L.: "Towards a Semantic-based Approach for Software Reusable Component Classification and Retrieval", Proceedings of the 42nd annual Southeast Regional Conference, Huntsville (2004)
30. Poulin, J.: "Populating Software Repositories: Incentives and Domain-Specific Software", Journal of Systems and Software, Vol. 30 (1995)