Towards Better Comparability of Software Retrieval Approaches Through a Standard Collection of Reusable Artifacts

Oliver Hummel, Werner Janjic
Software Engineering Group
University of Mannheim
Mannheim, Germany
{oliver, werner}@informatik.uni-mannheim.de

Abstract — **The idea of component-based software reuse as a cornerstone of a more engineering-like approach to software development has been around for more than four decades. Since software and its building blocks represent an important and valuable intellectual asset for most companies, researchers have been struggling for nearly the same time to get their hands on a substantial amount of reusable material to experiment with. Only the advent of the open source movement mitigated this problem considerably and hence inspired interesting new research in this area within the last decade. However, basically all novel software retrieval solutions of that period have been developed and evaluated independently from each other and are thus by no means comparable with one another. To address this flaw, an initiative was started to foster the creation of a reference reuse collection for software search and retrieval, which is intended as a common baseline for future comparison of software retrieval systems. In this paper we explain the motivation for this initiative, identify and discuss important foundations as well as open issues and present an initial sketch of architecture, content and practical prerequisites of such a collection.**

Keywords-component-based software development; software reuse; software search; software retrieval; reference collection.

I. Introduction

Despite the immense benefits that are attributed to the reuse of software [1] and a large number of seminal approaches (such as by Zaremski and Wing [2], [3]; see e.g. Mili et al. [4] for a comprehensive overview) developed in recent decades, Douglas McIlroy's initial idea of setting up market places with reusable components [5], [6] still has not lived up to its full potential [7]. Nevertheless, given today's exploding amount of potentially reusable (open source) software, freely available on the Internet, the need for effective software search and retrieval solutions – not only as an enabling factor for reuse – is more apparent than ever: open source repositories such as Sourceforge are hosting tens of thousands of software projects with millions of artifacts and even the version control systems of larger companies contain more files than a human can ever overlook.

Consequently, the so-called reuse repository problem [8] of not having enough material to fill repositories and market places with reusable components is no longer an issue since the Internet and the World Wide Web can be used as a source for harvesting reusable material [9]. However, in order to use this "megastore" of information for systematic software reuse, sufficiently sophisticated software retrieval approaches and tools are necessary. Especially when existing material was not initially intended for reuse (as is the case with most open source software today), this will only become accepted if developers are able to find and access useful components quick and easy. Consequently, this change of prerequisites has not only triggered a new wave of interesting academic research to better deal with search and retrieval of software artifacts (e.g. [10], [11], [12]), but has also created a new interest of commercial search engines (such as Koders, Krugle or formerly Google Codesearch) in searching for source code and software. Although all approaches available today are certainly important and have brought a new momentum to the community, they share one significant problem: to date, their evaluations, if existing at all, are largely based on different and/or proprietary datasets and thus it is impossible to objectively compare their performance on a common basis. Since even researchers are not able to assess the existing solutions and to understand their strengths and limitations, it is no surprise that software search and component reuse are still not widely adopted in industrial practice.

Interestingly, this evaluation challenge is not limited to component reuse alone; it is rather a problem that has been plaguing computer science (and especially software engineering) for a while. As observed by Tichy [13], computer scientists perform relatively little evaluations of their approaches so that the experimental paradigm is not as well established in our world as, for example in medicine or physics. In other words, computer scientists often focus too much on the development of new approaches and too little on their systematic evaluation, which makes it hard if not impossible to judge whether a new approach is really better than the previous ones. Certainly, the development of new approaches is important, but nevertheless, repeatable evaluations of new developments are at least as important for good research, as e.g. stressed by Basili [14] about twenty years ago: *"Proposing a model or building a tool is not enough. There must be some way of validating that the model or tool is an advance over current models or tools"*. In order to overcome this unsatisfying situation in the area of software search and retrieval, the creation of a reference collection of reusable artifacts was proposed recently [15]. The main motivation for this effort is to simplify the comparison of software retrieval systems. Furthermore, as already experienced in the text retrieval community such a collection

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-230-1

will offer a good starting point for the development of new and innovative tools as it freely provides the data necessary for initial experiments [16].

The remainder of this paper, discussing motivation and early experience from setting up such a collection, is organized as follows: First, we introduce the foundations of retrieval techniques and their evaluation, which is required to better understand the contribution of this paper and the usage scenarios in which the proposed reference collection can be used (Section 2). Subsequently, we briefly survey existing tool evaluations and identify their common weakness, before we shed some light on reference collections from related areas and how their ideas can be transferred to a standard collection of reusable components (Section 3). Section 4 introduces our approach for tackling this challenge and gives an example supposed to illustrate the usability of the approach before we conclude our paper in Section 5.

II. FOUNDATIONS

The origins of software search and retrieval can clearly be seen in “classic” text information retrieval [4] and therefore most early approaches for the former simply applied techniques from document retrieval to software artifacts (cf. [17]). Software retrieval, however, is potentially a far more complex undertaking than pure text retrieval since software does not only contain linguistic semantic information, but syntax and functional semantics as well. Zaremski and Wing were amongst the first researchers that elaborated on signature [2] and semantics specification [3] matching as a way of identifying reuse candidates. About ten years ago, Mili et al. [4] have presented a well-known survey that identifies five general groups of techniques applicable for the retrieval of software artifacts, namely –

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

The original listing contains a sixth group, called topological approaches, which from today’s point of view is rather an approach for the ranking of search results than a retrieval approach itself so that we have left it out in the enumeration. It obviously makes sense to reuse methods from information retrieval to perform simple textual analyses on software assets. Descriptive methods go one step further and require additional textual descriptions of the asset like a set of keyword or facet [18] definitions. Operational semantic methods rely on the execution or so-called sampling [19] of the assets. Denotational semantics methods use signatures (see e.g., [2]) or specifications [3] of artifacts for matching, while structural methods do not deal with the code of the assets directly, but with program patterns or designs. Overlap between these classifications can occur at various places, e.g., between (3), (4) and (5) as “behaviour sampling” [19] of components typically needs a specific signature to work on.

Based on the numerous results that had been presented in the late 1990s some researchers were even convinced that the most important software retrieval challenges have already been solved [20]. Existing prototypes were able to deal with the artifact collections available at that time easily (containing, however, often merely a few dozen elements). On the contrary, other researchers were convinced that the existing techniques would not be precise and usable enough when the amount of reusable material grows larger [4], which has been commonly seen as a required condition for successful marketplaces with reusable artifacts [7]. The latter assumption has at least preliminary been proven right almost ten years later when initial experiments [9] with “internet-scale” software collections have shown that the usage of merely one of the above mentioned retrieval techniques is usually not precise enough to deliver practically usable results. These experiments showed, e.g., that the precision of signature matching quickly drops to under one percent in collections with millions of artifacts. Consequently, in recent years, there has been an increasing interest in improving software retrieval approaches that led to a number of interesting approaches (as well as a number of high-profile publications [10], [11], [21]). Although their documentations include reasonable evaluations that demonstrate the prototypical applicability of the underlying approaches, it is impossible to compare them with each other as they were developed independently and evaluated with totally different methods and test collections. Even worse, the examples used to experiment with the prototypes are usually not publicly available and hence it is extremely difficult to judge the actual effectiveness of the evaluations and basically impossible to replicate the experiments performed.

Due to the conceptual proximity to information retrieval it is no surprise that common evaluation techniques from classic information retrieval are widely applied in the context of software retrieval. The two most prominent measures to assess the quality of retrieval systems are Precision P (measuring the fraction of relevant results D_r amongst all delivered results D) and Recall Re (the fraction of delivered relevant results D_r amongst all relevant results R):

$$P(D_r, D) = \frac{|D_r \cap D|}{|D|}$$

$$Re(D_r, R) = \frac{|D_r \cap D|}{|R|}$$

Further well-known but not so commonly used measures include Fallout (the fraction of non-relevant documents that is retrieved from all non-relevant documents) and the F1 measure (the weighted harmonic mean calculated from Precision and Recall) [16].

Recall is typically more important on small collections or on large collections with very specialized queries (where one assumes to have only few useful results per query), while Precision becomes more important on large collections with potentially numerous results. In this context, a tool should clearly minimize the amount of false positives since delivering only few relevant results amongst thousands of irrelevant candidates will not only result in a poor precision, but also in a low user satisfaction. It is obvious that such a

behavior is not tolerable in the area of software reuse where a careful assessment, selection and integration of potential reuse candidates may demand significant effort from the developer. Hence, a thorough assessment of a retrieval system typically requires the combination of at least two measures, since otherwise a system can be optimized for one and may fail in practice. The common approach to combine e.g., Recall and Precision in such evaluations is through so-called Recall/Precision curves [16] in which a large area under the curve indicates a well performing retrieval system. If other practical aspects are of interest other measures can be derived as well, including the search execution time of a query, for example.

A. Assessing Software Retrieval Tools

Although this general approach for the evaluation of software retrieval tools is undisputed, one aspect that complicates the evaluation is the challenge of defining the actual relevance of a reusable artifact. While determining the relevance of natural language documents is pretty straightforward for a human (e.g., does a document tell you how high Mount Everest is or not?), this task is much more challenging for software artifacts. As discussed before, the latter typically have three facets that can be used for retrieving them, namely linguistic information, the syntax of their interfaces and their semantics, i.e. their concrete functionality. As already observed by Mili et al. [4], the evaluation of software retrieval tools and algorithms is faced with a serious problem when it needs to find a good criterion that determines the practical relevance of a delivered result. Usually none of the three facets mentioned before is sufficient to achieve this on its own, as, e.g., text extracted from a component not necessarily describes its functionality in a precise and unambiguous manner; and even if a component with matching functionality has been found, a wrong interface might make its integration into a given environment hard or even impossible. In other words, a reusable component delivered by a state of the art software search engine might still require a significant amount of effort from a developer in order to finally determine whether it provides the desired functionality and is usable in the environment at hand. We believe, the ultimate *relevance criterion* for determining the reusability of a component and solving the make or buy dilemma [25] in favor of buy (reuse) is clearly the question whether a reusable artifact can be integrated into a system under development “as is”, i.e., with virtually “zero effort” and deliver the required functionality.

To our knowledge, however, this relevance criterion has rarely been consequently defined in the literature so far and thus, most previous evaluations have been relying on a kind of surrogate, namely the so-called *matching condition* that simply determines whether a search engine considers a document as relevant or not. Obviously, this does not reveal much useful information about the reusability of a component in a given context.

B. Usage Scenarios

Software development is a continuous and complex process that can benefit from software search at various occasions, which makes it important to identify and to bear in mind which usage scenarios exist for software retrieval tools within the software development lifecycle. Obviously, the process of “reusing” an artifact as an inspiration during the design or implementation phase of a software system is totally different to the actual reuse of a concrete component that needs to adhere to a given specification. While a stakeholder may be satisfied with relatively “blurry” results for the former, the latter requires a perfect match in order to make reuse more worthwhile than building the component from scratch, as explained before. Figure 1, taken from our earlier work [22], summarizes various archetypal usage scenarios for software retrieval systems and identifies the development activities where they are likely to be most useful. We used different shapes of lines, to illustrate distinction between more *speculative* (dashed) and *definitive* (solid) searches. The most important usages scenarios in the context of this paper are additionally highlighted in bold typeface.

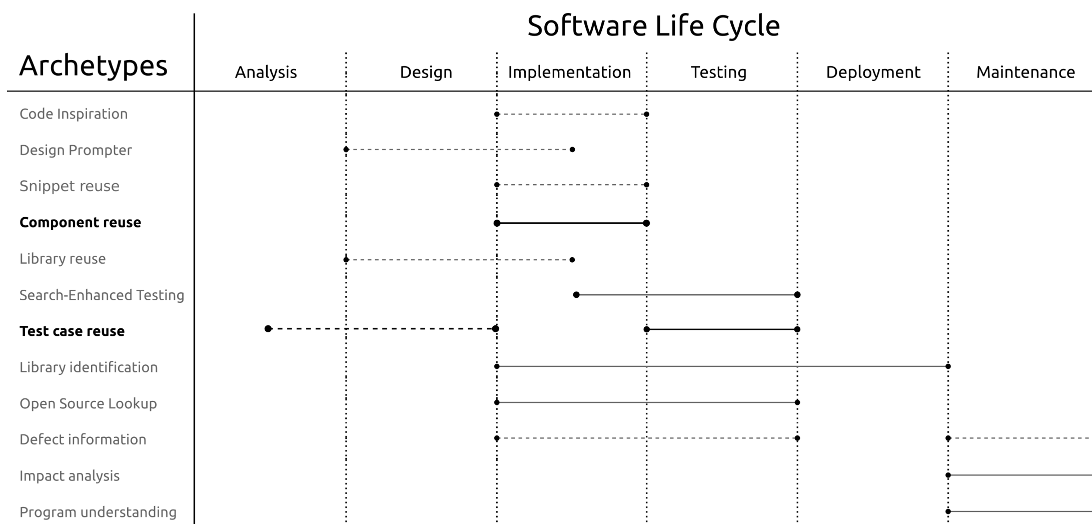

Figure 1. Overview of software retrieval usage scenarios and their possible times of application in the software development life cycle.

The motivation for definitive searches is always the concrete need for a specific artifact; let it be a reusable component described by a specification or a missing library that is required to overcome Java's infamous ClassNotFoundException. Both require a search engine to deliver as exact matches as possible and even near misses are usually not perceived helpful by the user, since they can often only be integrated with a substantial amount of modification, if they can be integrated at all. On the other hand where speculative searches are driven more by general information need, it is usually of interest whether any component or service is available for a given task at all. The exact shape of the desired artifact is typically not important in the context of such a speculative search, since the design of the application it should be integrated into is still moldable [23]. Due to a lack of space, we have to refer the interested reader to the original publication for more details on this topic; the focus of this paper and our initial work for a reference collection is on definitive searches used for the retrieval of well-defined reusable components.

III. SOFTWARE RETRIEVAL EVALUATIONS SO FAR

As indicated in the introduction, most software reuse approaches that have been published so far contain a reasonable evaluation that demonstrates their feasibility and leaves the interested reader at least with an idea of their potential and of potential problems. However, as can be seen in Table 1, that summarizes some of the best known reuse tools of the last 20 years, most of these evaluations (i.e., those that were performed on a component collection with more than just a few hundred elements) were incomplete from the perspective of classic information retrieval as they usually only calculated some kind of "top n Precision".

TABLE I. OVERVIEW OF EVALUATIONS OF PREVIOUS COMPONENT REUSE SYSTEMS.

Tool	**No. of Artefacts**	**Content**	**Input**	**Relevance Criterion**	**Measures**
Proteus [17]	~ 100	Unix commands	Keyword-based	Expert judgement	Precision, Recall, search time
CodeBroker [28]	~700	Java Classes	Signature and keywords	Expert opinions	Precision, Recall
SparsJ [10]	~180,000	Java Source Classes	Keywords	Expert opinions	Top n Precision
Maracatu [[29]]	~4,000	Java Source Classes	Keywords, facets	Expert's opinion (based on text matching)	Precision, Recall (only for subset of 200 artefacts)
Merobase [11]	~ 4 M	Java Source Classes	Test Cases	Passing of test cases	Top n Precision
Sourcerer [12]	~ 250,000	Java Source Classes	Keywords	Expert judgement	Hits per result page

This kind of "crippled" precision measure is typically used for search engines that operate on very large collections where it is not feasible to determine the relevance of all (potentially thousands of) results that may be returned for a query. Instead, human experts revise only the, e.g., 20 highest ranked results (n = 20) for their relevance. Further results and the Recall (for which knowledge of all relevant elements in a collection is required) are simply ignored. This procedure is usually justified by the habit of human users of internet-scale (commercial) search engines that typically do not consider more than roughly the first 20 results. However, for a scientific comparison of search engines this is obviously neither sufficient nor satisfying, especially in the area of software retrieval where both, high precision and high recall are essential as explained before.

A. *Reference Collections so far*

Tool evaluations in computing are often challenging, as they typically require expensive empirical investigations to demonstrate that a tool is better than other tools available before [14]. However, software engineering is certainly not the only discipline in computer science that has to deal with somewhat fuzzy requirements to its tools. Therefore, the idea of creating reference collections that allow benchmarking of tools is certainly not new. Take, for example, the "Siemens Testing Suite" [30], a popular collection of programs containing known errors, which was widely used during the 1990s to evaluate the effectiveness of test cases and test case creation strategies. More specifically, the challenge of evaluating retrieval approaches is clearly known in related disciplines as well. First and foremost, it is certainly the information retrieval (IR) community [16] that found itself in trouble how to evaluate their emerging text retrieval algorithms some twenty years ago. At that time there were a lot of new and exiting ideas as well as prototypes around in this community, but the proprietary (and often very expensive) evaluations performed on them individually were usually not very helpful and especially not comparable with each other. Fortunately, the IR community was able to overcome this challenge by defining so-called reference collections comprising a large set of documents, a substantial number of tasks for retrieval systems and the expected solutions for them. The most prominent one is probably the Text REtrieval Collection (TREC) [16] that has been considered as a major success fostering IR research since its creation tremendously. Although TREC as a text-based collection is not of direct use for the retrieval of software artifacts, it can still be used to learn about some basic principles how to define and built such a reference collection. Furthermore, in the long term, the results gained with

Copyright (c) IARIA, 2012.

it might also be helpful by giving some insights on heuristics that can help to improve text-based retrieval algorithms for software retrieval tools with techniques such as stemming or the use of thesauri [16].

A second group that has been struggling with the comparability of its tools is the rather young community trying to match and orchestrate (semantic) web services. As it is dealing with executable artifacts as well, it is obviously more closely related to the retrieval of software components than pure text retrieval. Given the enormous amount of money that for instance was recently spent by the European union (Küster and König [24] talk about 70 million Euros) to support the research on semantic web services, it is not a surprise that especially European researchers came up with the idea of setting up a reference collection of semantic services to evaluate matching tools and have been driving this idea ever since. The so-called S3 (for Semantic Service Selection) collection is the initial result of these endeavors. The current version of S3 contains 1.083 semantically (with 38 ontologies) annotated web services and a set of 42 queries for them. Various participants of the S3 contests and related workshops have manually identified services in the collection they considered relevant for each query in order to create a set of relevant answers. To our knowledge together with the OPPOSUM portal [24] (that subsumes S3 and a few significantly smaller collections) it forms the only baseline that allows systematic comparison of (ontology-based) software retrieval algorithms so far. To our knowledge, there exists no similar undertaking for a specific reference collection in the reuse area for the time being.

Limitations of Web Service Reference Collections

However, although this can also be seen as a first step towards a better evaluation of software retrieval algorithms, its applicability in the context of software reuse is questionable for a number of reasons. First and foremost, the introduction of a graded relevance scheme and the revision of the relevant results in the 2010 version of the S3 collection changed the perception of relevance considerably and it seems that there is still a large degree of subjective judgment that influences the understanding of relevance here. Thus, the risk that even this sophisticated collection does not contain a clear notion of relevance, as we demanded it for the evaluation of software reuse tools, is high. Second, most of the existing software retrieval and reuse approaches operate on source code, which is by definition not available from web services, while vice versa, source code available in open source repositories is usually not annotated with any kind of ontological information. Moreover, the size of the existing S3 collection is still rather limited (compared with current software reuse collections as introduced in Table 1) and the chance of substantially increasing it seems low, as the definition of relevant results and the annotation of the indexed services with ontological information is effort manual activity. Given the size of state of the art reuse collections that already goes into the millions, it is not clear whether the results obtained from such a small collection can be scaled up to internet-scale search engines. Finally, the current S3 collection is focused on speculative searches that are supposed to deliver all services that can be (remotely) helpful for a query; the actual syntax of a service is currently not taken into account. In other words, a definitive match between query and result is highly unlikely in this collection and a composition of various results may be required to create a service that is finally able to satisfy a concrete request.

B. Requirements for a Reuse Reference Collection

Küster and König [24] identified a number of desirable characteristics for a semantic web service collection in their publication and obviously it makes sense to revisit their work as a starting point for a reference collection for software reuse. In total they list the following five major points:

1. Expressivity & Usability: contained elements need to be described as precisely as possible in order to avoid room for interpretation of the results.
2. Scope: the collection should comprise elements from as many different domains as possible in order to maintain a high diversity and to allow making statements, which approaches work under which circumstances.
3. Scalability & Size: since large testbeds are required to properly evaluate retrieval approaches, the collection must be kept scalable.
4. Automation: obviously, the use of the collection should be automated as much as possible.
5. Decoupling: as many people as possible should contribute to the endeavor in order to avoid unintended bias in the collection.

In general we can accept this list of requirements as helpful for a reuse reference collection as well, although requirements 1) and 3) are clearly contradicting each other in the context of a very large collection. This fact makes a precise relevance criterion even more important because it is not possible to manually investigate millions of artifacts for their relevance. But nevertheless, it is important to preserve as much information as possible when content for the collection is harvested, as different usage scenarios for software search engines may require slightly different information to evaluate the retrieval algorithms.

Special Requirements in the context of Component Reuse.

As discussed before, the main motivation for the use of a component collection from a reuse point of view is to find a concrete artifact that definitively fills an existing gap. Besides other factors, it has been mentioned numerous times in the literature [25] that a reusable component must be large enough so that reusing it is cheaper and easier than self-implementing it (often called the "make or buy decision"). Otherwise the incentive for a developer to reuse is obviously low. While a component was initially seen as function by McIllroy [5] in his seminal reuse paper, the granularity of components has continuously been growing since then and today a component is typically seen as an independently deployable part of a system [6], comprising numerous classes (if developed in an object-oriented language) behind

a well-defined interface. At the same time, research has started to investigate automated adaptation [26] of components and automated orchestration [24] of (usually semantically annotated) services or in other words: automated "glue coding". As the use of clever glue coding is likely to increase the haul of matching components from a collection (and thus to influence the number of relevant components) in the future, we believe it makes sense to consider the following three categories of automated glue coding when competing search engines are to be evaluated in the context of a reuse reference collection:

1. No glue coding at all: only direct matches are allowed, no changes beyond simple path and package configurations can be made to reuse candidates.
2. Adaptational glue coding: adapters [27] that wrap a single component in a 1:1 fashion (or change them internally) are allowed.
3. Compositional glue coding: the 1:n orchestration of multiple sub-components behind a newly created interface in the sense of the facade pattern [27] is allowed.

IV. PROPOSED APPROACH

The two most important "ingredients" for a reuse reference collection are certainly a large collection of reusable material and a large enough collection of (at least some) non-trivial queries that can be used to challenge search engines and is not under the suspicion of being biased for a particular engine. Moreover, a good way of determining the relevance of retrieved candidates needs to be found. Since we have already faced this challenge during the evaluation of our Merobase search engine [11], we believe a good starting point for this is the collection of test cases (e.g., written in JUnit) that can be used to doubtlessly judge whether a delivered result is relevant or not. Ideally, a search engine would directly support the use of such test cases for automating this assessment, as Merobase does, for example. The technique behind such a feature is known in the literature as test-driven reuse [11] [21]. The following two subsections go into more detail on this before we present the results of an exemplary query that demonstrates the practical usability of our approach and conclude this section with a brief discussion of our preliminary findings.

A. Data Sets

In the context of the ICSE workshop on Search-driven development: Users, Infrastructure, Tools and Evaluation (SUITE) in 2010 a working group was formed with the goal to evaluate the feasibility of creating a reuse reference collection. As a result, the groups of Christina Lopez at the University of California in Irvine and our group at the University Mannheim have agreed to make the collections forming the backbone of the software search engines Sourcerer ([12], http://sourcerer.ics.uci.edu) respectively Merobase ([11], http://merobase.com) available on the Web so that they can be downloaded via http://resuite.org. Currently, several hundred gigabytes of data are available there and hence processing and indexing these collections is certainly a matter of weeks if not months: Irvine's collection comprises about 500,000 java source files from roughly 13,000 open source projects, while our collection consists of about 3 million java files harvested from nearly 50,000 open source projects. To our knowledge these two packages form the largest body of open source software freely available today. Given its large size, it is likely that it will not only be facilitating experiments for software reuse, but in related areas (such as the community organizing the Mining Software Repositories conference) as well.

B. Queries

As briefly mentioned before, we plan to start the creation of queries for the reference collection based upon the test cases we created as input for evaluating our earlier work [8] to which we have to refer the reader for further details due to the limited space of this paper. To our knowledge, test cases are currently the best available technique that allows formulating a semantically precise and automatically checkable specification for reusable software components. A further advantage of test cases is that they can easily be "translated" into input for other retrieval approaches as well. Consider the following simple JUnit test case that is supposed to test an equally simple Stack data structure:

```
public class StackTest extends TestCase {
     public void testStack() {
       Stack s = new Stack();
       assertTrue(s.isEmpty());
       s.push((Object)"Object1");
       s.push((Object)"Object2");
       s.push((Object)"Object3");
       assertFalse(s.isEmpty());
       assertEquals(s.pop(), (Object)"Object3");
       assertEquals(s.pop(), (Object)"Object2");
       assertEquals(s.pop(), (Object)"Object1");
       assertTrue(s.isEmpty());
     }
   }
```

From this piece of code it is, for example, possible to extract keywords (such as stack, push, pop, isEmpty) or the complete interface of a stack required to satisfy this test case without much ado. Moreover, even the extraction of a simplified description of the Stack's behavior is contained in this test case. In addition to the above mentioned set of test cases we are aware of two other recent publications that used test cases (or at least test data) for a similar purpose and contain further evaluation challenges (cf. [21] & [12]). We have recently made all test cases that have been used for test-driven reuse available as JUnit test cases via resuite.org as another pillar for the reference reuse collection described in this paper. Since all three approaches are currently in a prototypical stage, there is no precision recall analysis available. Nevertheless, a sufficient quality of the test cases guarantees that retrieved candidates are able to deliver the desired functionality. Reusable components that have actually been retrieved beyond simple data structures such as stacks or binary trees, include a validator for credit card numbers, spreadsheet calculation and Blackjack logic, a comprehensive overview can be found in the mentioned publications ([11], [12], [21]).

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-230-1

C. Examplary Results

Based on the test case we just introduced, we carried out an exemplary analysis of various retrieval techniques in order to show the expressivity improvement that we were able to achieve in comparison to the simple top 20 precision determination used in earlier evaluations [11]. In particular, we analyzed interface-based (only classes with identical interfaces, i.e. all names, parameter and return types had to match), name-based (only names of classes and methods had to match) and signature-based (only the parameter and return types had to match, names were ignored [2]) searches for their Recall and Precision as shown in the following figure:

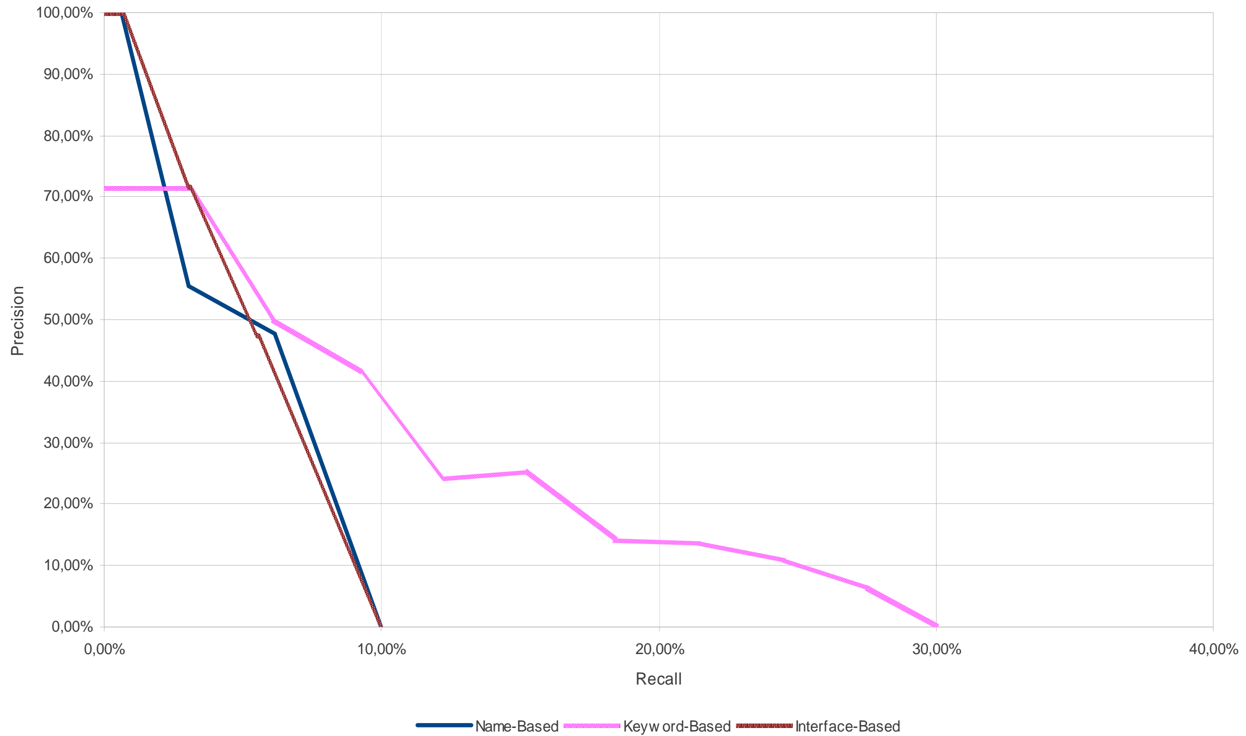

Figure 2: Recall/Precision curve of different retrieval algorithms for the Stack test case from above.

In total, the signature required by the above test case yielded a pool of 454,541 classes from the Merobase collection that at least contained the three method signatures defined and thus theoretically had the potential for being usable as stacks. In practice, only a small fraction of them – namely 163 – have been successfully tested and delivered this functionality with the current version of our testing tool (supporting adaptation and rudimentary dependency resolution, but no composition). Thus, 163 was used to calculate Recall and Precision. As visible in Figure 2, the relatively simple retrieval algorithms used for this experiment suffer from either a low recall or a low precision as summarized in the subsequent table.

TABLE II. COMPARISON OF RETRIEVAL ALGORITHMS.

	Name	Interface	Signature	Keyword
Max. Recall	8.0 %	5.5 %	100 %	28.2 %
Precision at max. Recall	50.0 %	47.4 %	< 0.1 %	5.8 %
No. Relevant / Candidates	13 / 36	9 / 19	163 / 454,541	46 / 3,000

D. Discussion and Forthcoming Steps

The approach we have just described already forms a useful core for a reference collection of reusable artifacts. Our preliminary results indicate that an evaluation based on such a collection with results known as relevant is feasible for internet-scale software repositories as well and delivers significantly better results than the top n precisions usually calculated for such repositories. However, as long as only one tool with potentially imperfect adaptation has been used to identify the relevant results for a query, it is not sure that all relevant results have actually been discovered. Only a combination of various tools and approaches can guarantee a (nearly) perfect coverage of relevant results and thus create a valid baseline for the calculation of Recall and Precision.

Therefore, one central prerequisite for the creation of a viable reference collection is to have a large body of researchers and working groups contributing their ideas and tools. We would like to invite the community to challenge, discuss and extend the requirements and the contents of the collection in its current state. Although contact requests via email are always welcome, we believe it makes sense to discuss the further proceeding personally with as many

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-230-1

people as possible and hence plan to organize a workshop on this topic soon. Another important future step is the creation of a web portal (similar to the one described by Küster and König [24] for semantic web services) that offers easy access to data sets and queries.

E. Open Questions

Compared with a simple text reference collection or even a web service collection, a generic reuse collection is faced with a number of additional questions we briefly need to mention at this occasion.

Although web service descriptions are by definition programming language independent, the elements of a collection of reusable components, however, can be created in any arbitrary programming language. In other words, their evaluation would require support for test-driven reuse in each of these languages. For the time being only three different prototypes of a test-driven reuse system in the Java programming language exist. Clearly, it makes sense to set up similar collections for other languages in order to study whether a different language will affect the performance of retrieval algorithms in any way.

Another issue closely related with the programming language is the question whether an artifact is compilable and executable at all. Often source files have dependencies on other source files and will not be testable without either complex dependency resolution algorithms as available in the Eclipse framework and used by e.g., Code Genie [12] or without the complete metadata (build path, etc.) of the original project. Since most software search engines today still focus on individual classes (cf. table 1) we rely on the simple dependency resolution mechanisms contained in our tool right now and bear in mind that they are not perfect. Hence, it is likely that other tools might discover additional relevant results in the future through the use of better dependency resolution. However, a similar progression of relevant results has been observed during the creation of the TREC collection, so that this is perfectly acceptable.

The TREC collection has another advantage over a software reference collection, namely the one that texts that are once written (such as newspaper or research articles) typically are not changed later. However, in the context of software retrieval it is very likely that the projects forming the collection will be updated over time and hence the question arises whether and how updates can be performed. Updating the collection itself is essentially noncritical as it just calls for replacing, adding or removing files; the actual challenge is to identify all results that may have become relevant or irrelevant after such an update.

A final issue to deal with is the question how to decide what makes a component or project elevated enough to become part of a reference collection? We are well aware that one may allege a certain bias for elements included e.g., in our Merobase collection so far. However, we believe that the sheer size of about 3 million Java source files is already large enough to mitigate such allegations. Furthermore, it only contains open source projects harvested from popular open source hosters (such as SourceForge) and no specifically tailored projects that would harden this suspicion. Moreover, the idea is that a future collection is extensible so that everyone interested is able to contribute his own material to it.

V. Conclusion and Future Work

Component-based software reuse is by no means a new concept and as widely demonstrated in the literature a sophisticated software retrieval tool is an essential building block to make reuse work. However, despite decades of intensive research and the significant progress made in software retrieval in recent years, it is still hard to compare existing reuse approaches, as there exists no common testbed for this purpose. As we have discussed in this paper, recent efforts to set up a semantic web service reference collection are certainly a step in the right direction, however, since the prerequisites and goals of this community are different to those of the component reuse community, it is unlikely that results gained with this collection can be transferred to software component retrieval.

Thus, we have proposed to create a reference collection with reusable components based upon two recently published collections of files from more than 50,000 open source projects. Our proposal includes creating definitive queries for concrete reusable artifacts in the form of test cases that can be used to determine free of doubt whether a delivered candidate will be usable in a given context specified by the test case. Such test cases may be seen as a rather harsh relevance criterion for reusable software components, but ultimately they are the only way to establish the fitness for purpose of a component and in our understanding this is the only way to lower the threshold currently still hindering systematic reuse in practice. Moreover, we defined three classes of adaptation approaches that may be used to classify the contestants that should be compared with a reuse reference collection.

Such a collection will not only be applicable for comparing existing tools with full Recall / Precision curves, it is also likely that it will simplify the creation of and initial experimentation with other innovative tools in the future. Furthermore, there is a high chance that the data sets will be useful for other communities (such as the one that is mining software repositories, for example) as well and hence we invite researchers from all related areas to contribute to the efforts in setting up this collection as well.

References

[1] Krueger, C.W.: Software Reuse, ACM Computing Surveys, Vol. 24, Iss. 2, 1992.

[2] Zaremski, A.M. and Wing, J.M.: Signature Matching: A Tool for Using Software Libraries. ACM Transactions on Software Engineering and Methodology, Vol. 4, Iss. 2, 1995.

[3] Zaremski, A.M and Wing, J.M.: Specification Matching of Software Components, ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 4, 1997.

[4] Mili, A., Mili, R., and Mittermeir, R.: A Survey of Software Reuse Libraries. Annals of Software Engineering 5, 1998.

[5] McIlroy, D.: Mass-Produced Software Components. Software Engineering: Report of a Conference sponsored by the NATO Science Committee, Garmisch, Germany, 1969.

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-230-1

[6] Szyperski, C.: Component Software, 2nd Edition, Addison-Wesley, Amsterdam, 2002.

[7] Ravichandran, T. and Rothenberger, A.: Software reuse strategies and component markets. In Communications of the ACM, 46, 8, 2003.

[8] Hummel, O.: Semantic Component Retrieval in Software Engineering, PhD Dissertation, University of Mannheim, Germany, 2008.

[9] Hummel, O. and Atkinson, C.: Using the Web as a Reuse Repository. In: Morisio, M. (ed.) Proceedings of the International Conference on Software Reuse, LNCS 4039, Springer, Heidelberg, 2006.

[10] Inoue, K., Yokomori, R., Fujiwara, H., Yamamoto, T., Matsushita, M., and Kusumoto, S.: Ranking Significance of Software Components Based on Use Relations. IEEE Transactions on Software Eng., Vol. 31, Iss. 3, 2005.

[11] Hummel, O., Janjic, W., and Atkinson, C.: Code Conjurer: Pulling Reusable Software out of Thin Air. IEEE Software, Vol. 25, Iss. 5, 2008.

[12] Bajracharya, S., Ossher, J. and Lopes, C.: Sourcerer: An internet-scale software repository. Int. Workshop on Search-Driven Development, SUITE 2009.

[13] Tichy, W.: Should computer scientists experiment more? IEEE Computer, Iss. 5, 2002.

[14] Basili, V.: The Experimental Paradigm in Software Engineering. Experimental Software Engineering Issues: Critical Assessment and Future Directions, Springer, 1993.

[15] Hummel, O.: Facilitating the Comparison of Software Retrieval Systems through a Reference Reuse Collection. Int. Workshop on Search-Driven Development, SUITE 2010.

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

[17] Frakes, W.B. and Pole, T.P.: An Empirical Study of Representation Methods for Reusable Software Components. IEEE Transactions on Software Engineering Vol. 20, Iss. 8, 1994.

[18] Prieto-Díaz, R.: Implementing faceted classification for software reuse. Communications of the ACM, Volume 34, Issue 5, 1991.

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

[20] Poulin, J.: Reuse: Been There, Done That. Commun. of the ACM, Vol. 42, Iss. 5, 1999.

[21] Reiss, S.P.: Semantics-based Code Search. Int. Conf. on Software Engineering, 2009.

[22] Janjic, W., Hummel, O., and Atkinson, C.: More Archetypal Usage Scenarios for Software Search Engines. Int. Workshop on Search-Driven Development, SUITE 2010.

[23] Crnkovic, I., Chaudron, M., and Larsson, S.: Component-based Development Process and Component Lifecycle, Proc. of the Intern. Conf. on Software Engineering Advances, 2006.

[24] Küster, U. and König-Ries, B.: Towards standard test collections for the empirical evaluation of semantic web service approaches. Int. Journal Semantic Computing, Vol. 2, Iss. 3, 2008.

[25] Clements, P.: From Subroutines to Subsystems: Component-Based Software Development. in Heineman, G., Councill, W. (eds..: Component-based Software. Eng. Ad.-Wesley, 2001.

[26] Hummel, O. and Atkinson, C.: Automated Creation and Assessment of Component Adapters with Test Cases. Intern. Symposium on Component-Based Software Engineering, 2010.

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

[28] Ye, Y. and Fischer, G.: Reuse-Conducive Development Environments. Journal of Automated Software Engineering, Vol. 12, No. 2, Kluwer, 2005.

[29] Garcia, V., Lucrédio, D., Durão, F., Santos, E., Almeida, E., Fortes, R., and Meira, S.: From Specification to Experimentation: A Software Component Search Engine Architecture, International Symposium on Component-Based Software Engineering, CBSE 2006.

[30] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T.: Experiments on the effectiveness of dataflow- and control flow-based test adequacy criteria. International Conference on Software Engineering, 1994.

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-230-1