Chapter 12
Test-Driven Reuse: Key to Improving Precision of Search Engines for Software Reuse

Oliver Hummel and Werner Janjic

Abstract The applicability of software reuse approaches in practice has long suffered from a lack of reusable material, but this situation has changed virtually over night: the rise of the open source movement has made millions of software artifacts available on the Internet. Suddenly, the existing (largely text-based) software search solutions did not suffer from a lack of reusable material anymore, but rather from a lack of precision as a query now might return thousands of potential results. In a reuse context, however, precisely matching results are the key for integrating reusable material into a given environment with as little effort as possible. Therefore a better way for formulating and executing queries is a core requirement for a broad application of software search and reuse. Inspired by the recent trend towards test-first software development approaches, we found test cases being a practical vehicle for reuse-driven software retrieval and developed a test-driven code search system utilizing simple unit tests as semantic descriptions of desired artifacts. In this chapter we describe our approach and present an evaluation that underlines its superior precision when it comes to retrieving reusable artifacts.

12.1 Introduction

Building high-quality software faster, cheaper, and more predictable with the help of reusable software building blocks is not a new idea. The earliest publication usually referenced in this context is Douglas McIlroy's seminal paper on component reuse [25] presented at the famous NATO conference in Garmisch in 1968 where amongst other ideas the term "software engineering" was coined. The idea of "remixing"

O. Hummel (✉) • W. Janjic
Software Engineering Group, University of Mannheim, 68131, Mannheim, Germany
e-mail: hummel@informatik.uni-mannheim.de; werner@informatik.uni-mannheim.de

existing software parts in order to compose new systems [24] as one cornerstone of a more engineering like software development approach [27] certainly fitted well into the spirit of the whole event.

Software engineering research in general has come a long way since then and has successfully identified a lot of reuse potential amongst various software engineering artifacts [9]. The systematic reuse of well-defined third-party software building blocks (such as components [6] or services [7]) according to a well-defined specification (as e.g. envisaged in [1]) is one of the most challenging approaches since it requires a precise matching of potential reuse candidates to a given specification. However, most existing software search solutions are still text-based and do neither reflect nor support the need to match reuse candidates with the syntactic and semantic characteristics of a specification. Nevertheless, the programmatic syntax and semantics make software search and retrieval significantly different from plain text retrieval so that the techniques that have been successfully applied within the information retrieval community are likely not to be sufficient in the context of (reuse-driven) software retrieval. Software retrieval research has identified some core challenges for implementing a sustainable reuse repository that requires to –

- Create and maintain a large enough software collection that makes searches promising (the so-called repository problem [35]),
- Index and represent its content in a way that makes it easily accessible (the representation problem [8])
- Allow characterizing a desired artifact with reasonable effort and precision (the usability problem [10])
- Execute queries with high precision in order to retrieve the desired content (the retrieval problem [32]).

While the open source movement, higher bandwidths and always increasing hardware power seem to have mitigated the repository problem recently (cf. Sect. 5.2.2 as well), the other three challenges center around finding an optimal representation of software artifacts that allows storing, retrieving and searching them in a precise manner with little effort.

In the remainder of this chapter we present a practical solution to tackle all three of these challenges. We especially describe how we increased the precision of software searches from a reuse perspective based on ordinary unit tests as usually created during every software development project anyway [4]. We have found that well formulated test cases reveal enough syntactical information and semantics of a desired component that they can be used as a query for software searches effectively. Therefore, the current status of our test-driven reuse approach, first presented in 2004 [15], is described in detail in Sect. 12.3. Moreover, we show how we integrated this vision in a state of the art software search engine and into the developer's work environment through a plug-in for the popular Eclipse IDE. In Sect. 12.4 following thereafter, we explain how we evaluated our system in order to demonstrate the feasibility of the test-driven reuse approach and compare it with a similar system recently presented in the literature. Section 12.5 describes this and other related work in more detail before we share our view of the most important open challenges in the context of reuse-oriented retrieval and conclude our chapter in Sects. 12.6 and 12.7, respectively.

12.2 Foundations

Around the turn of the millennium, the state of the art in reuse-oriented software retrieval could be characterized from two widely contrary viewpoints. While one opinion was that most challenges related with reuse libraries have been solved already since repositories were supposed to be mere catalogs containing only a small number of about 50 to perhaps 250 carefully selected components [31], the other opinion claimed the exact opposite. Mili et al. realized in their well-known survey on software retrieval approaches [26] that there is indeed a large variety of prototypical component library systems, but none of them would be able to overcome the retrieval and usability problem mentioned in the introduction, as soon as the amount of available components would increase considerably. The reason for this pessimistic appraisal is simple: since the *matching criterion* (such as e.g. the appearance of a keyword anywhere – i.e. even in comments – within a candidate) is rather weak in most approaches, it is very likely that they return many irrelevant results as soon as a critical mass of indexed material has been exceeded. In other words, 10 years ago, it was still unclear how to build *internet-scale* software search systems (with potentially millions of entries) that would be able to deliver only those components precisely matching an existing "gap" in an application under development.

Separating the useful from the useless results in growing collections is one of the major challenges for all information retrieval approaches [3] and is usually referred to as the *precision* of a search engine. More formally, precision is defined as the fraction of relevant results amongst all results returned for a query. Obviously, it becomes tedious to determine the actual relevance of more than perhaps a few dozen results so that evaluations of common search engines often limit themselves on investigating the precision of the first 20 results (the so-called top 20 precision [23]). A similar challenge exists for the second central metric used in information retrieval to evaluate search engines, the so-called *recall*, defined as the fraction of all relevant elements that are returned for a given query out of all relevant elements contained in a collection for that query. As a collection may actually contain thousands of relevant results, it is often not possible to determine all of them and therefore the recall cannot be determined as well.

The results of a systematic survey in which Mili et al. [26] analyzed existing software retrieval solutions for their performance are presented and discussed in the following. The authors were able to distinguish six seminal classes of software retrieval approaches that we will briefly explain after their enumeration. However, due to a lack of access to most prototypes and the non-existence of standardized evaluation scenarios, Mili et al. were only able to estimate the potential performance (i.e. recall and precision) of these approaches on larger collections so that interested readers are referred to their publication for further details. The estimates they published are as follows:

1. Information retrieval methods (Recall: high/Precision: medium)
2. Descriptive methods (Recall: high/Precision: high)
3. Denotational semantics methods (Recall: high/Precision: very high)
4. Operational semantics methods (Recall: high/Precision: very high)
5. Structural methods (Recall: very high/Precision: very high)
6. Topological methods (Recall: unknown[1]/Precision: unknown)

Software retrieval is a specialization of *information retrieval* and hence it makes sense to reuse methods from the latter area to perform a simple, purely text-based retrieval of software assets. *Descriptive methods* go a small step further and rely on external textual descriptions (i.e. metadata) for an asset. Hence, Mili et al. denote such descriptive methods as a subset of the information retrieval methods, but due to the high use of this approach in practice and literature they created an additional category. *Denotational semantics methods* use signatures (see e.g. [42]) or formal specifications [43] of the indexed assets for retrieval. While signature matching is widely seen as a practical tool in this context, as it uses the parameters and return values exhibited in the interface of an artifact for matching, software retrieval based upon the matching of formal specifications suffers from a variety of disadvantages (such as difficulties in creating and evaluating them). *Operational semantics approaches* that rely on the execution of the indexed software with sample input values are certainly expensive to execute, however, they seem to be easily automatable. Nevertheless, also appealing in theory, this approach definitively also comes with some practical challenges: side effects, non-termination, the structure of used data types, dependencies, etc. can cause serious problems. Hence, in this context, it is no surprise that the most well-known implementation so far, called Behavior Sampling [30], was merely applied to simple mathematical functions of the C standard library. *Structural methods* finally do not deal with the code of the assets directly, but rather with internal program patterns or designs. Since it is largely unclear how to formulate queries for such an approach, it does not surprise that it has only rarely been experimented with.

Overlap between the discussed classifications can appear at various places, e.g. between (3) and (4) and (5) as the "sampling" of components typically needs a specific signature or structure to work with. As visible in the list, Mili et al. still defined *topological methods* as an independent class of approaches, however, since their common denominator is the *distance* between the query and the candidates, we would prefer to describe it as an approach for ranking search results that can (exclusively) be used together with at least one concrete instance of the other approaches.

[1] For topological methods it is difficult to define or estimate recall and precision. See [26] for more.

12.2.1 State of the Art

As previously indicated, the premises for software search and retrieval have changed considerably since the Internet and its users have made millions of open source components [16] available for software developers and researchers alike in recent years. Around the turn of the millennium, Seacord realized this potential and attempted to fill a software repository with Java applets collected on the Web by an automated crawler [36]. Also at that time, Ye was one of the first researchers recognizing that software searches are not only hindered by technical weaknesses of search engines, but by usability issues as well. He found that developers are often not even aware of the chance that a reuse candidate might be stored in a repository (which was understandable though due to their relatively small size at that time) and hence proposed and implemented a prototypical software search system (called CodeBroker) that continuously monitors the work of a developer and proactively presents potentially reusable candidates based on textual information from the comments athe developer has been writing [41].

Also around that time, the World Wide Web witnessed the rise of large-scale search engines helping to make its growing amount of data accessible. Inspired by the success of Google's PageRank algorithm [29], it was the ComponentRank approach of Inoue et al. [22] that breathed new life into the software retrieval community with an automated search engine (known as Spars-J). While their basic retrieval approach was still text-based and hence simple, it was their set of about 150,000 open source files that was far larger than every other collection before, together with the clever ranking approach that created a new standard. Inoue et al. proposed to rank those components higher in the result list of a search that are more often used than others amongst the indexed files. Nevertheless, the overall precision of the searches remains still too low from a specification-based reuse perspective as long as merely keyword matching is applied. Almost simultaneously, Hummel and Atkinson [16] demonstrated that general web search engines (such as Google) could be used for software searches by enriching queries with special keywords (such as: filetype:java AND "class stack") that – though not working absolutely perfectly – still delivers relevant source code with a high hit ratio.

However, although all seminal search approaches described before were available at that time, little work is known that would have tried to integrate them with the upcoming large-scale software search engines described in the next subsection. Consequently, a pure text-based retrieval still remained state of the art at that time. The only visible progress was the idea of parsing source codes in order to extract the names of objects and their methods to allow more focused searches for them (as e.g. introduced by Koders.com). Hummel et al. have coined the term *name-based* retrieval for that technique [17]. Retrieval approaches such as signature matching [42] or interface-based retrieval – the combination of signature and name-based retrieval (also described in [17]) – did not find their way into any of this new generation of software search engines. Numerous of them have been developed during the last 10 years and a good number is still available on the World Wide Web. As demonstrated by the various software search engines that have been launched as well as shut down in recent years, operating such an engine is not per se a fast-selling item, it rather seems to be related with a considerable risk to receive a lack of interest when content and usability are not appealing enough to potential users. The prime example in this context is certainly the failed high-profile attempt of IBM, Microsoft and SAP to establish the so-called UDDI Business Registry (UBR) as a marketplace for (web) services that was finally closed down in early 2006 containing barely a few hundred entries of dubious quality [16]. However, even operating a popular search engine does not guarantee its long-time survival as is underlined by the recent announcement of Google to shut down its code search engine in January 2012 [13].

In spite of that, various code search engines (academic as well as commercial) have demonstrated that the advances in database and text search technology (such as the Lucene framework [14]) have made the creation of "internet-scale" software repositories a viable undertaking wherefore the *repository problem* can be regarded as solved. In order to conclude this subsection, the following table summarizes important characteristics of some of these second generation software search engines.

Table 12.1: Overview of code search engines and directories

Name	Year	No. of artifacts	Retrieval algorithms	Remarks
UDDI Bus. Reg.	2000	<500 services	Keyword matching on metadata	Shutdown in 2006
Spars-J	2004	$>10^5$	Keyword matching	Implements ComponentRank
Koders.com	2004	$>3\cdot10^9$ LOC	Keyword & name matching	Commercial by Black Duck SW
Google Codesearch	2006	$>10^7$	Keyword matching/regular expressions	Shutdown January, 15 2012
Sourcerer	2007	$>10^6$	Keyword & name matching	Eclipse integration via CodeGenie plug-in

A more comprehensive overview that demonstrates even more forcefully that top notch software search engines today are easily able to index millions of artifacts can be found in [19] and in another chapter of this book [5].

12.2.2 Remaining Challenges

From the four problems identified for reuse-driven software retrieval in Sect. 12.1, state of the art software search engines thus have basically solved the *repository problem* and the *representation problem* by creating *internet-scale* collections of software assets that can be managed with common databases or state of the art search frameworks such as the freely available Lucene [14]. However, the *usability* and the *retrieval problem* dealing with how to efficiently retrieve the artifacts that are useful in a given context are still in the focus of interest in the research community. Garcia et al. [10] have recently underlined this with their list of requirements for a component search engine: amongst other challenges they see a simple query formulation and a good retrieval quality at the heart of a successful and scalable component search engine. Unfortunately, as has been shown recently [17], simple keyword- or signature-driven searches may lead to tens of thousands of results from which – in principle – each one matches the given query criterion. However, only because a – in these two cases – relatively simple technical matching criterion is fulfilled, a search result does not necessarily become relevant for the user (see e.g. [26]). Consider, for example, that a search for a reusable "spreadsheet" component merely delivers a test case for a spreadsheet because it naturally has to use a spreadsheet and thus contains the term. A user presented with such a result would certainly be disappointed and after inspecting perhaps five or ten similar results not consider using the search engine again, as in a reuse context, it is important to get results precisely matching a given specification [1].

Interestingly, most existing software search engines still rely on a simple keyword matching so that they suffer from exactly this problem. Although it seems possible to narrow down the search results considerably through adding more keywords to a certain degree, beyond that there still existed no intuitive approach for formulating interface-based or even specification-based queries in second generation software search engines as described in another chapter of this book [5]. Only Google's code search engine allowed the use of (rather complex) regular expressions in order to describe the desired interface of a component.

12.3 Test-Driven Reuse

According to the classification of Mili et al. presented in Sect. 12.2, the test-driven reuse approach Hummel and Atkinson have first introduced in 2004 [15], is a technique based on operational semantics and hence inspired by the ideas of Behavior Sampling by Podgurski and Pierce [30]. Due to their random nature, Behavior Sampling requires a relatively large number of samples even for simple functions and, to our knowledge, was never used in practice.

What is extensively used in practice, on the contrary, is (or at least should be) systematic software testing with targeted "samples" of a software's functionality derived with the help of some systematic approach such as equivalence class partitioning. In case of so-called test-driven development [4], which is especially popular in agile development communities, test cases are even created before any production code is written and are used to monitor the production code's degree of completeness and correctness during development iterations. From this starting point it is just a small step to imagine the usefulness of test cases in determining the fitness for purpose of reuse candidate. Assume as an example that we need a component offering the functionality of a typical spreadsheet application (such as Excel), i.e., it should be able to organize cells and reference them in alphanumerical form (rows as numbers, columns as alphabetic characters), hold values in these cells, use them within formulas and hence allow calculations based on other cells' values. A simple JUnit 3 [4] test for such a functionality might look as the one depicted in Listing 12.1. It describes two things, namely first, the (rather brief) required interface of the Spreadsheet component as shown by the UML class in Fig. 12.1 and second a concrete description of the required functionality, against which potential reuse candidates can be tested. Although the interface of this component is simple, it obviously requires quite some code to manage the cells of a spreadsheet and to parse and evaluate their contents.

Based on the above test case and the interface of the required component "hidden" within it, a search can be issued to an arbitrary software search engine. As soon as results are delivered from there, it should be possible to compile and test them against the JUnit test case. Whenever the test case can be compiled and executed successfully against a reuse candidate, it can be assumed that a working implementation for the specified functionality has been found. Figure 12.2 summarizes this "test-driven reuse cycle" as initially introduced in [15].

As depicted in Fig. 12.2, it is also possible to fully automate this cycle: a developer merely needs to specify the tests (step a) and then waits until the system delivers successfully tested reuse candidates (step f). The steps b to e in between can be automatically executed by an appropriate reuse system (we will explain a

Listing 12.1 JUnit test case testing (and hence describing) a simple spreadsheet component.

```
public class SpreadsheetTest extends TestCase {
    private Spreadsheet sheet;
    public void setUp() {
        sheet = new Spreadsheet();
        sheet.put("A1", "5");
    }
    public void testCellReference() {
        sheet.put("A2", "=A1");
        assertEquals("5", sheet.get("A2"));
    }
    public void testCellChangePropagates() {
        sheet.put("A2", "=A1");
        sheet.put("A1", "10");
        assertEquals("10", sheet.get("A2"));
    }
    public void testFormulaCalculation() {
        sheet.put("A2", "3");
        sheet.put("B1", "=A1*(A1-A2)+A2/3");
        assertEquals("11", sheet.get("B1"));
    }
}
```

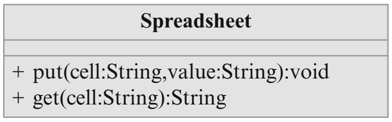
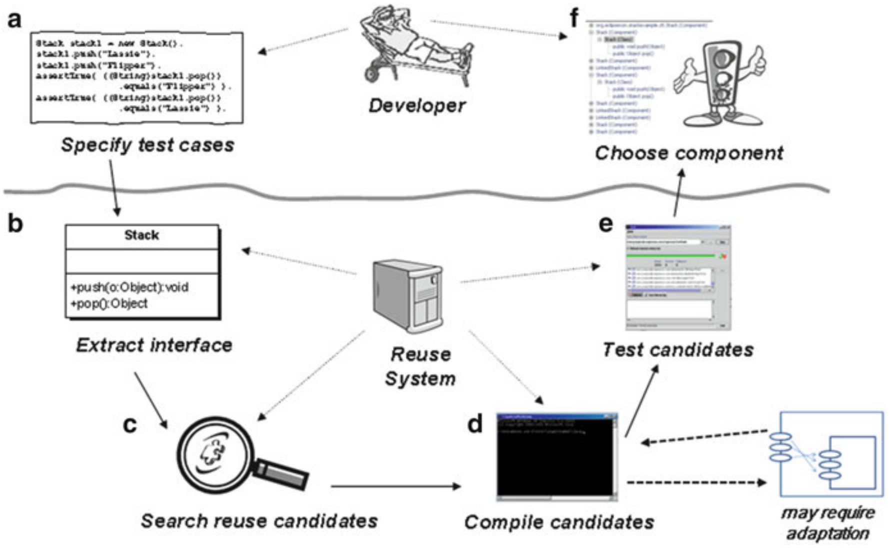

Spreadsheet
+ put(cell:String,value:String):void
+ get(cell:String):String

Fig. 12.1 The interface of a simple spreadsheet component as defined by the test case in Listing 12.1

concrete implementation in the next subsection). In step b the required interface of the desired reuse candidate is extracted from the test case what leads to step c where a query is derived to drive an arbitrary code search engine (we use our Merobase search engine that is explained in more detail in Sect. 12.3.1). While in principle it would also be feasible (if not easier) to search and test binary files, our current implementation still focuses on source code files (because historically there used to be little support for searching binary files in software search engines) that need to get compiled against the test case in step d and in case this procedure was successful are tested in step e.



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

Fig. 12.2 The test-driven reuse "cycle"

Figure 12.2 furthermore shows that the desired interface from the test case is not always fully matched by a potential reuse candidate when it comes to compilation in step d. As a matter of fact, it is not exactly matched in most of the cases so that the test case will not compile out of the box with most results, which is of course unsatisfying. As a way out of this dilemma, we have been developing a test-based adapter [11] generator that is able to automatically "wrap" the reuse candidate with the appropriate interface in order to make it compilable and executable. In principle, it simply creates all syntactically possible adapter "wirings" and selects the one that successfully passes the specified test case, implementation details and a proof of concept implementation can be found in another publication [21].

Feeding the example test from Listing 12.1 to Merobase, eventually yields four successfully tested reuse candidates. One of them is particularly interesting as it nicely demonstrates how a "real component"[2] consisting of various classes (as illustrated in the UML diagram from Fig. 12.3) can be discovered with our approach through merely describing and testing its facade [11].

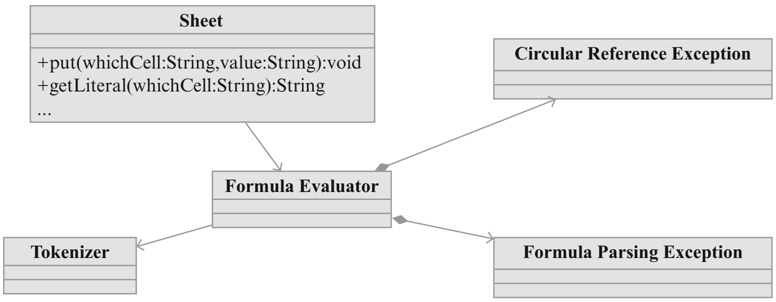

Sheet
+put(whichCell:String,value:String):void
+getLiteral(whichCell:String):String
...
Circular Reference Exception
Formula Evaluator
Tokenizer
Formula Parsing Exception

Fig. 12.3 Simplified structure of retrieved spreadsheet component as UML class diagram

The class ensemble discovered here consists of three main classes, enclosing two inner exception classes, and comprises in total slightly more than 300 lines of code. Obviously, the interface of the facade class does not match the interface specified by the test case from Listing 12.1 and thus would not compile. Hence, our tool created the adapter presented in Listing 12.2 that provides exactly the interface required by the test case and forwards requests to the retrieved component.

12.3.1 Implementation

Garcia et al. [10] depict the necessity of integrating source code search into the IDE of the developer, as this prevents a loss of concentration and a media-break for switching to another tool (like e.g. a web browser). Of course, reuse-oriented IDE plug-ins usually cannot work as standalone tools, but must connect to a software repository server via the Internet. In this section, we explain how our group at the University of Mannheim has tackled this challenge and describe our software search engine Merobase and its associated Eclipse plug-in Code Conjurer [20]. While Merobase distinguishes itself from other software search engines through its broad support of retrieval techniques, Code Conjurer is able to deliver proactive reuse recommendations by silently monitoring a user's work (i.e. the code a developer writes in Eclipse) and triggering searches automatically whenever this seems reasonable. We

[2] Result source (visited Dec, 14th 2011): http://www.purpletech.com/xp/wake/src/Sheet.java

Listing 12.2 Automatically generated adapter for the Sheet result.

```
public class Spreadsheet {
   private adaptee.Sheet adaptee;

   public Spreadsheet() {
      adaptee = new adaptee.Sheet();
   }

   public String get(String whichCell) {
      try {
         return adaptee.get(whichCell);
      } catch (RuntimeException e) {
         if (e instanceof RuntimeException) {
            throw e;
         }
         return null;
      }
   }

   public void put(String whichCell, String value) {
      try {
         adaptee.put(whichCell, value);
      } catch (RuntimeException e) {
         if (e instanceof RuntimeException) {
            throw e;
         }
      }
   }
}
```

will explain this process in more detail in Sect. 12.3.1.2 Potentially reusable results are shown in Code Conjurer's recommendations view (cf. Sect. 12.3.1.2) where we explain our tool in more detail. Figure 12.4 describes the overall process in our reuse recommendation system, including our Merobase search engine, the Code Conjurer plug-in and the virtual machines used for secure testing of retrieved candidates.

12.3.1.1 Merobase: A Search Engine Supporting Test-Driven Reuse

The index creation for our Merobase repository is driven by automated crawlers that can harvest source and binary files from three different sources, namely CVS and SVN repositories as well as from websites (via HTTP). While the repository crawling requires a list of projects to download the files from the respective (open source) repository, web crawling works with an extended version of Lucene's Nutch crawler [14] starting from some seed URLs. The index itself is also based on the

Lucene framework [14] and currently contains about ten million files from well-known open source hosting sites and the open Web (roughly 8 %), out of which roughly 40 % are binary files (primarily Java archives, but some .NET binaries as well). Special parsers for each supported programming language allow to extract syntactical information, store it in the index and search for it later. In addition to class and method names, we store operation signatures (i.e. parameter and return types) and complete operation headers (i.e., operation signatures plus names) as concatenated terms optimized for Lucene in the index. Details on their structure can be found in another chapter of this book [5]. Currently, Merobase is able to work with Java, C++ and C# sources, WSDL files, binary Java classes from Java archives (JARs) and .NET binaries.

Whenever a user sends a request to the Merobase server (either through the web-interface available at merobase.com or a client program like Code Conjurer accessing its web service based API), the above parsers and a special JUnit parser (able to extract the interface of the class under test from test cases) are invoked and try to extract as much syntactic information from the query as possible. If none of the parsers recognizes parsable code, however, a simple keyword search is executed. Based on parsed syntactic information, Merobase supports retrieval by class and operation names, signature matching and by matching the full interface of classes as described before. Although preliminary results indicate that the latter indeed leads to a higher precision with common "toy examples" [17] collected from the literature, the risk of "over-specifying" desired components is certainly also real, as e.g. the previous spreadsheet example has demonstrated: no candidate completely matched the relatively simple interface we have specified. Nevertheless, the retrieved components that were finally working successfully, were found amongst roughly 22,000 results of a "relaxed" query that merely searched for the desired signatures (i.e. ignored class and operation names in the interface). As searches for more complex interfaces often tend to deliver few results (as e.g. predicted by Crnkovic [6]), we have integrated a number of strategies into Merobase for relaxing queries as well. Further details on the index structure of Merobase, its content, and the applied matching strategies are explained in another chapter of this book [5].

In case of a test-driven search, which is triggered when a JUnit test case (such as the one in Listing 12.1) is submitted, Merobase automatically tries to compile, adapt and test the highest ranked candidates. If a candidate is relying on additional classes, the algorithm uses dependency information to locate them as well (as seen in the spreadsheet example). As visible in Fig. 12.4, the actual compilation and testing are not carried out on the search server itself, but on dedicated virtual machines within sandboxes. These ensure that the executed code does not have the possibility to do anything harmful to the user's system or bring the whole testing-environment down; in our publicly available system we have also deactivated network transfer to prevent abuse. Another system continuously monitors the virtual machines (by polling a special monitoring service provided by the sandboxes) and as soon as it recognizes that one is not working properly, it simply replaces it with a new instance, which takes about 30 s for replacing and restarting.

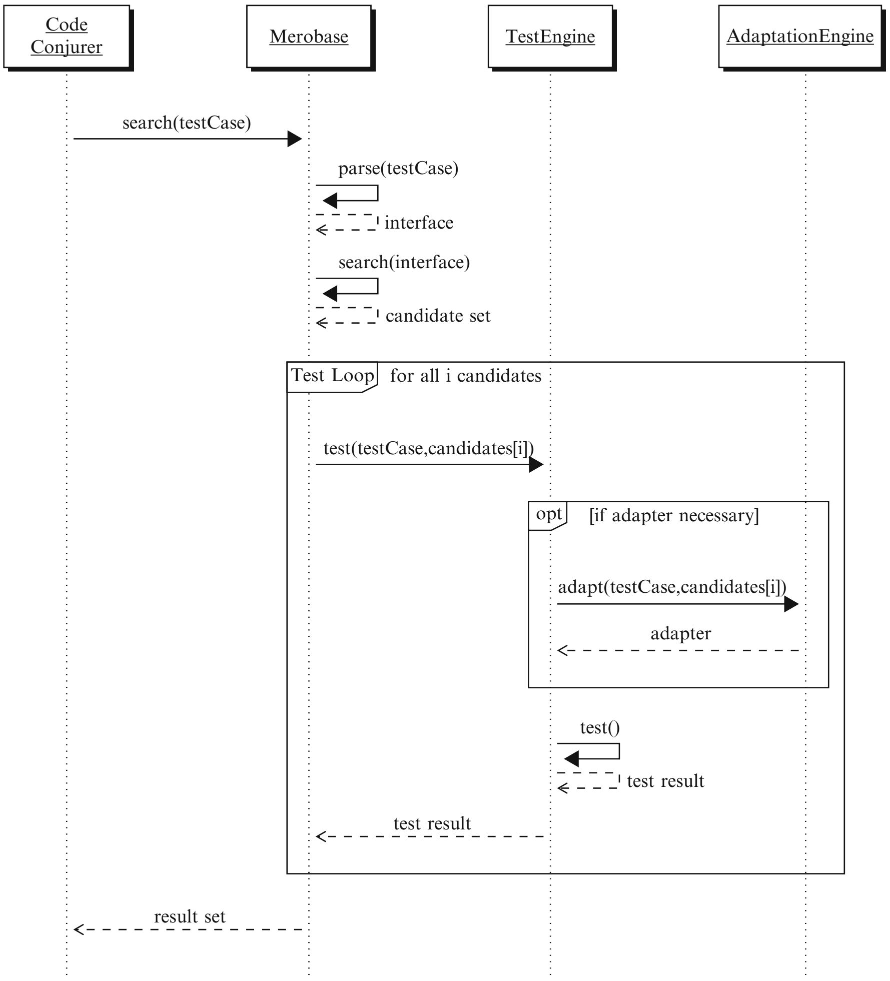


Fig. 12.4 Architecture for a test-driven software reuse environment

12.3.1.2 Code Conjurer: Test-Driven Reuse in Eclipse

Although the Merobase search engine is certainly a useful tool, its regular web interface forces a developer to leave his development environment when he wants to search for reusable artifacts. Even worse this requires the cognitive decision that reuse is desired in a particular situation [41], which clearly disturbs the creative thought process of software development. Thus, we have created the open source tool Code Conjurer[3] as a plug-in for the Eclipse IDE, installable through the Eclipse

[3] Which is hosted on sourceforge.net and available at www.code-conjurer.org

Marketplace. It integrates Merobase's interface-driven searches as well as the test-driven search technology into a widely used development environment.

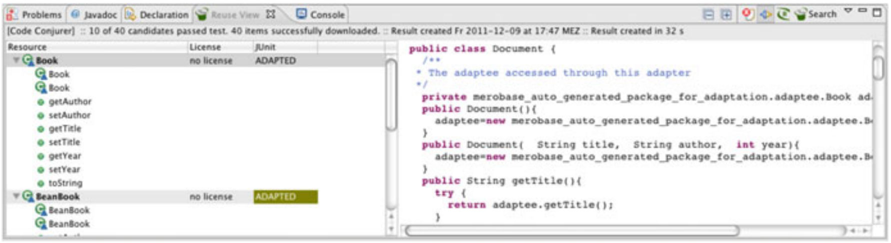
After Code Conjurer is installed, it presents itself with a small magic hat icon and a reuse menu in the Eclipse toolbar. The default position of the reuse view as visible in Fig. 12.5 is at the bottom of the workbench where it presents all necessary information about reusable assets and performed searches. Code Conjurer neither requires the user to learn any dedicated query language nor to consciously write any queries at all. When activated, it simply extracts the queries from the current source window a developer is working with. Since Code Conjurer focuses on Java, Java classes and JUnit 3 test cases[4] are supported as queries in this context.

For interface-base searches, Code Conjurer assists the developer with a non-intrusive background agent, that searches for reusable artifacts. The algorithm judging when a search should be triggered is developed continuously and actually relies most on changes of the interface description of the class under development. Hence, Code Conjurer triggers background searches whenever a method is added to the class, deleted or its signature is changed. Search results are presented in a tree view, while next to them a code preview is offered for the selected item.

If the user decides to enable the test-driven search feature of Code Conjurer, the background agent monitors changes to the interface of the so-called *class under test* (CUT), which is – in contrast to the provided interface of the class in interface-base search – the required interface of the JUnit test case written by the developer. This approach is very close to the one propagated by Extreme Programming, which encourages the developers to iteratively write tests that fail, then implement the desired functionality and then add more tests that fail again. In our case, the developers would not implement functionality, but reuse existing software assets.

When executing test-driven searches, Code Conjurer sends the JUnit test to Merobase where the interface of the desired class is extracted from the code and used to search for results. Retrieved candidates are distributed to special virtual machines used for compilation and testing. After the execution of the tests against the candidates, the test results are shown in Code Conjurer's result view from where they can be directly added to the working project via drag and drop. The example shown in Fig. 12.5, shows the results of a test-driven search. They all required an adapter to work with the provided test case (which is shown by a bar in yellow ochre. When a result is chosen for reuse, the adapter is automatically integrated into the working project along with the reusable class. In other words, the retrieved code is directly usable and the test initially defined by the user can then be executed locally on the retrieved code, in order to ensure that it has been integrated correctly.

[4] The only requirement is that the tests should be written according to best SE practices (e.g. the name of the test should reflect the class under test's name).



[Code Conjurer] :: 10 of 40 candidates passed test. 40 items successfully downloaded. :: Result created Fr 2011-12-09 at 17:47 MEZ :: Result created in 32 s

Resource	License	JUnit
Book	no license	ADAPTED
Book		
Book		
getAuthor		
setAuthor		
getTitle		
setTitle		
getYear		
setYear		
toString		
BeanBook	no license	ADAPTED
BeanBook		
BeanBook		

```
public class Document {
  /**
 * The adaptee accessed through this adapter
 */
  private merobase_auto_generated_package_for_adaptation.adaptee.Book ad
  public Document(){
    adaptee=new merobase_auto_generated_package_for_adaptation.adaptee.B
  }
  public Document(  String title,  String author,  int year){
    adaptee=new merobase_auto_generated_package_for_adaptation.adaptee.B
  }
  public String getTitle(){
    try {
      return adaptee.getTitle();
    }
```

Fig. 12.5 Code Conjurer's "window to the world": the reuse view with a generated adapter

12.4 Evaluation

As the information retrieval community has experienced in its early years, objectively evaluating retrieval tools is difficult [3]: as mentioned before, information retrieval science envisages the usage of recall and precision to judge the usefulness of an information retrieval tool. However, it is well accepted that the exact recall of large-scale search engines cannot be determined easily, since it is usually not possible to identify all relevant candidates from a corpus with millions of entries. Text retrieval reference collections that have been built with lots of effort hence normally are using manually inspected retrieval results collected from various tools to establish a baseline for the recall, unfortunately, those collections do not (yet) exist for software retrieval [20]. Another issue with test-driven search is that its precision is by definition equal to 100 % (assuming the test cases are expressive enough) since retrieved candidates can be directly integrated into the project under development. Hence, we believe a first reasonable evaluation of test-driven reuse approaches is to demonstrate its feasibility by applying it to a variety of search challenges found in literature. Since recently another implementation of the approach has become available [33] and presented a similar evaluation we also compared its results with our system in order to further underline the technical feasibility of the overall approach.

12.4.1 Assessing Reuse Challenges

For the evaluation of our approach we have collected a number of previously published retrieval challenges from related literature and as far as necessary created simple JUnit test cases (documented in their entirety elsewhere [18] due to their size). We have used our stable initial test-driven reuse prototype and the Merobase search engine to search for Java source codes and to execute the test cases. Since this prototype is still testing all candidates sequentially and identifies feasible adapters based on a brute-force approach [21] the times shown in the following table can be seen as absolute worst cases for testing all potential candidates with a matching signature. The comparison following in the next subsection already demonstrates that more user-friendly search times of under 3 min can be achieved by parallelization and optimization of adapter creation and of course by incremental result delivery.

The results are summarized in Table 12.2 that contains the interface specified in the respective test case its first column. Columns two and three compare interface-based retrieval where candidates have to match this interface exactly (including class and operation names) with a signature-based retrieval where it is sufficient when a counterpart with the matching parameters and return types can be found for each specified method. We also show the number of components that passed the test vs. the total number of candidates found by Merobase in each cell, e.g. for the interface-based retrieval of a Stack, 150 components out of 692 candidates were able to pass the test. The numbers below indicates how much time the prototype required to try out all candidates retrieved by Merobase. Finally, the last column lists "synonymous" class names we have found amongst the successfully tested candidates using (more relaxed) signature-based retrieval.

Table 12.2: Overview of successfully solved reuse challenges

Query	Interface-based	Signature-based	Exemplary result classes for signature-based harvesting
`Stack ( push(Object):void pop():Object )`	150/692 26 min 45 s	611/35,634 18 h 23 min	`Stack, MyStack, ObjectStack, Keller, LIFO, Pila, ObjectPool, LifoSet`
`Calculator ( sub(int,int):int add(int,int):int mult(int,int):int div(int,int):int )`	1/4 19 s	22/23,759 20 h 24 min	`Calculator, CalculatorImpl, Molecule, Arithmetic, SimpleMath, Operators`
`Matrix ( Matrix(int,int) get(int.int):double set(int,int, double):void multiply(Matrix):Matrix )`	2/10 26 s	26/137 5 min 25 s	`Matrix`
`ShoppingCart ( getItemCount():int getBalance():double addItem(Product):void empty():void removeIt(Product):void )`	4/4 26 s	4/12 47 s	`ShoppingCart`

Query	Interface	Signature	Exemplary results

`Spreadsheet (`
`  put(String,String):void`
`  get(String):String`
`)`
	0/0
	3 s
		4/22,705
		15 h 13 min
			`Sheet,`
			`Compiler,`
			`Util`

`ComplexNumber (`
`  ComplexNumber(double,`
`    double)`
`  add(ComplexNumber):`
`    ComplexNumber`
`  getRealPart():double`
`  getImagineryPart():`
`    double`
`)`
	0/1
	3 s
		32/89
		1 min 19 s
			`ComplexNumber`

`MortgageCalculator (`
`  setRate(double):void`
`  setPrincipal(double):`
`    void`
`  setYears(int):void`
`  getMonthlyPayment():`
`    double`
`)`
	0/0
	4 s
		15/4,265
		3 h 19 min
			`Loan,`
			`LoanCalculator,`
			`Mortgage`

On the one hand, the results presented in the table demonstrate the capability of the test-driven reuse approach as we were able to identify a number of artifacts in our collection that are able to deliver quite complex functionalities (such as a Spreadsheet or a Matrix) as specified in the test cases. On the other hand it also demonstrates its largest two dilemmas, namely the problem of "over-specifying" the desired artifact and the execution time. The more complex an interface becomes, the harder it gets to find a perfectly matching implementation. Although, relaxing the search criteria indeed increases the probability of success, it increases the time required for testing so that it is still difficult to apply test-driven reuse in practice where developers demand results within just a few seconds. In principle, however, distributing the testing to a large number of virtual machines should decreases this time significantly as is shown in the subsection following hereafter.

12.4.2 Comparison

Recently, the idea of test-driven software search has been adapted by Steven Reiss [33] from Brown University with his tool S6 and by Lemos et al. at UC Irvine with their Eclipse plugin CodeGenie [39]. We will give some more details on their approaches in the following section on related work, but first we want to demonstrate that our implementation is able to reproduce results similar to those reported by Reiss. Unfortunately, at the time of writing, CodeGenie required triggering the testing of each candidate manually within Eclipse so that we were unable to include it into the comparison for reasons of security and effort.

Table 12.3 presents the results of our comparison in five columns starting with a reference to the used example. Columns two and three illustrate how many successfully tested results have been discovered by Merobase within the first 500 candidates (using its "relaxed" search approach described before) without respectively with automatic adaptation of mismatching signatures. The JUnit test cases we used were created according to the test samples provided by Reiss in his paper [33], his results are reproduced in the fourth column for a direct comparison, while the fifth column is reserved for special remarks where necessary. Due to the limited number of candidates and optimizations in terms of parallelization and adaptation, this time Merobase required less than 3 min per example, which certainly seems a reasonable number for practical use. This time we have executed the testing in a parallelized environment (running on an AMD Opteron based server with a 2.6 GHz dual-core processor and eight virtual machines) that yielded results comparable with Reiss's system that required between 15 and 169 s in a testing environment utilizing also eight threads.

Table 12.3: Comparison of test-driven search implementations

Example	Merobase unadapted	Merobase adapted	S^6 [33]	Remarks
SimpleTokenizer	0	2	14/138	
QuoteTokenizer	0	0	4/6	
Robots	–	–	1/124	Not repeatable[5]
Log2	0	1	1/100	
FromRoman	0	2	3/38	
ToRoman	2	4	6/56	
Prime	0	4	14/228	
PerfectNumbers	0	1	5/28	
DayOfWeek	0	0	0/89	3/5,000
Easter	0	0	1/6	Not repeatable[6]
MultiMap	0	0	2/165	3/10,000
UnionFind	0	2	1/149	
TextDelta	0	7	1/249	

[5] Reiss' tests required resources from the Web that are not available anymore.

[6] We were not able to find results with Reiss' tool either.

The remarks for *DayOfWeek* and *MultiMap* in Table 12.3 were Merobase could not find results within the first 500 candidates mean that it was able to discover three working version in a larger set of candidates (5,000 resp. 10,000). However, the expressiveness of this comparison is unfortunately still somewhat limited since Reiss has used different search engines with different retrieval algorithms that finally delivered different candidates. It nevertheless demonstrates that Merobase achieves a similar performance as another contemporary tool and is also able to deal with completely unbiased reuse challenges independently specified by someone else so that the technical feasibility of test-driven reuse has been illustrated one more time.

12.5 Related Work

After some years of relative silence around the turn of the millennium, a new momentum has become visible in the software retrieval community in recent years and other approaches implementing a test-driven reuse approach have been presented by other researchers. To our knowledge, two research groups have been developing and experimenting with appropriate tools. As already mentioned, Reiss has developed S6 [33], a web-based search tool where a user can list search keywords, specify the declaration of one or more method headers and add test samples that describe the semantics of the desired operation. According to Reiss's publication, S6 is also able to "adapt" retrieved candidates by carrying out various internal program transformations based on the abstract syntax tree of the potential result and to retrieve numerous operations within one Java class. S6 is able to use its own search engine called Labrador or a number of other code search engines such as Koders or Sourcerer.

Sourcerer itself, which was developed by Bajracharya et al. [37] at the University of California in Irvine implements a ranking approach similar to ComponentRank [22] and is the foundation for another test-driven reuse tool called CodeGenie [39]. In contrast to S6, and similar to Code Conjurer, CodeGenie is fully integrated into the Eclipse IDE and able to directly use JUnit test cases to drive a search for a missing Java method. In order to do so, CodeGenie analyses Eclipse's compiler errors and tries to find missing classes respectively their methods via the Sourcerer search engine. The user can inspect the candidates delivered by Sourcerer and can request from CodeGenie to "weave" them into his project where they can be tested as usual with the help of JUnit. One of the main contributions of Sourcerer and CodeGenie is probably their ability to work even with declaratively incomplete program files (so-called slices) that can also be woven into the project under development. In contrast to Code Conjurer that always integrates complete files and tries to resolve missing dependencies also on a per file basis, CodeGenie thus seems to be more flexible, as far as this can be determined without a direct comparison on the same data set. Clearly, it would be interesting to see such a comparison (of course also including the capabilities of S6) to better understand the advantages and disadvantages of all three tools, however, this is yet to be done.

Other recent approaches for increasing the precision of software searches in large-scale repositories include the work of Grechanik et al. [12] who have built a search engine that analyses the documentation of API calls (e.g. Javadocs) with common information retrieval approaches in order to retrieve complete applications that implement a desired high-level functionality (such as "record midi file"). It thus avoids the need for exactly matching components and adaptation in the first place. In terms of size of their search target, a number of innovative tools such as XSnippet [34] or ParseWeb [38] reside at the other end of the spectrum as they mainly support developers in Eclipse through finding examples for object instantiations and API calls. However, we are currently not aware of any other recent approaches that also aim on retrieving reusable software building blocks according to a concrete specification as test-driven reuse does.

12.6 Future Work

Although test-driven reuse marks another milestone for specification-based software search and retrieval, there still exist many aspects with potential for improvement as already illustrated by the three currently available approaches [19, 33, 39] with their individual strengths and limitations. Since they only support the reuse and integration of Java source code so far, it is certainly interesting transfer the approach to other programming languages, although we do not see any reason why this would cause major problems. In order to make test-driven reuse applicable for the daily work of a developer, however, it is necessary to further decrease the time until result are delivered. This can of course be done by a further parallelization of test execution (with corresponding costs, of course), or by improving the adaptation generation and of course by optimizing the underlying search engine so that it simply ranks potentially working results higher. Moreover, many advanced techniques from information retrieval such as stemming, synonyms or hypernyms [3] have occasionally been tried out for software search, but not yet systematically investigated so far so that their effects are not yet clear. However, this problem has been plaguing general information retrieval systems for years: for example, naively adding synonyms as search terms, quickly leads to an explosion in the number of results and in turn most likely to decreasing precision. Thus, one goal for the near future should be the discovery of an optimal mix of heuristics that delivers an acceptable amount of tested results within a reasonable amount of time. In other words, we still need to find out which software retrieval algorithm works best for which usage scenario, as there is still a lack of systematic evaluations as recently criticized [20].

Another challenging but not less important question is, whether and how complex class ensembles or, in more general terms, complex components can be best retrieved in a widely object-oriented world. There, today's mainstream applications are mostly composed of very fine-grained building blocks (i.e. the classes) and thus composing an object-oriented program with the test-driven reuse approach would in principle require a detailed specification for each desired object. On the one hand, creating each class individually is what needs to be done in object-oriented software development anyway, however, on the other hand it clearly contradicts the idea of composing preferably large components and hence defeats most of the benefits of component-based development [40] that hides implementation details behind interfaces. "Carving" components from a bunch of objects currently only works automatically as long as a hull (better known as a facade [11]), such as the Sheet class from Figure and Listing 12.1, is coincidentally available and all its dependencies can be resolved. To our knowledge, automated orchestration mechanisms as they are intensively investigated in the web service community (e.g. in [28]) are not yet supported by any of the current software (or service) search engines and prototypes. A prerequisite for overcoming this challenge is of course being able to find all dependencies a reuse candidate relies upon. While we have already discussed some simple heuristics for this task, a systematic analysis of this area is also still open.

12.7 Conclusion

The contributions we have described in this chapter are manifold, we have presented a novel approach hat uses ordinary (unit) test cases for search and retrieval of well defined software building blocks in a reuse context. We have described the current state of development of our proactive reuse recommendation tool and a search engine that can be used to implement test-driven reuse in practice. Furthermore, we have applied our tool to a number of realistic reuse challenges demonstrating that the approach is technically feasible, which is also supported by two similar implementations published recently. Moreover, we have identified some interesting ideas for improvement and once more realized that it is about time to carry out a systematic comparison of (not only test-driven) software search tools, based on a unified reference collection.

Since testing still is (and will certainly remain for some time to come) the only means by which software components can be judged as "fit for purpose", we believe that, together with a test-driven reuse approach, it can become the central driver for component and service markets in the mid-term future. Thus, our basic idea is to integrate the ability of testing components and services into future versions of software brokers (such as the former UDDI Business Registry). In addition to delivering components that syntactically match users queries, search engines enhanced in this way will also be able to execute tests in order to filter out those reuse candidates that are not fit for the desired purpose.

In contrast with current testing approaches, however, a new form of "blind testing" is required to protect the interests of component providers and users in such a commercial brokerage scenario. Thus, we propose a novel form of testing in which a search engine only provides the user with an indication of whether a test was passed or failed, but not with the actual results delivered by the component under test. Moreover, it is important that the expected result of a test submitted by a user is also not disclosed to the component since it could otherwise be used to return spoofed results that might influence a purchase decision [2]. Thus, search engines in our future concept need to act act as a trusted broker between component providers and potential users (i.e. buyers). This vision certainly has the potential to bring the practice of software reuse closer to McIlroy's long-felt desire of viable software component marketplaces.

Acknowledgements The authors would like to thank Colin Atkinson, Philipp Bostan, Daniel Brenner, Matthias Gutheil, Christian Ritter, Marcus Schumacher and Dietmar Stoll from the Software Engineering Group at the University of Mannheim for their contributions to developing the tools described in this chapter. Furthermore, we would like to express our gratitude for the helpful comments of the anonymous reviewers.

References

[1] 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)

[2] Atkinson, C., Brenner, D., Hummel, O., Stoll, D.: A Trustable Brokerage Solution for Component and Service Markets. Proceedings of the Intern. Conference on Software Reuse (2008)

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

[4] Beck, K.: Test-driven development: by example. Addison-Wesley (2003)

[5] Hummel, O., Atkinson, C., Schumacher, M.: Artifact Representation Techniques for Large-Scale Software Search Engines. In Sim and Gallardo-Valencia (eds.): Finding Source Code on the Web for Remix and Reuse, Springer, 2012.

[6] Crnkovic, I.: Component-based software engineering – new challenges in software development. Software Focus, Vol. 2, No. 4 (2001)

[7] Erl, T: Service-Oriented Architecture: Concepts, Technology and Design. Pearson (2005)

[8] Frakes, W.B.: An empirical study of representation methods for reusable software components. IEEE Transactions on Software Engineering, Vol. 20, no.8 (1994)

[9] Frakes, W.B., Terry, C.: Software Reuse: Metrics and Models. ACM Computing Surveys, Vol. 28, No. 2 (1996)

[10] Garcia, V.C., de Almeida, E.S., Lisboa, L.B., Martins, A.C., Meira, S.R.L., Lucredio, D., de M. Fortes, R.P.: Toward a Code Search Engine Based on the State-of-Art and Practice. Proceedings of the Asia Pacific Software Engineering Conference (2006)

[11] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software, Addison-Wesley (1995)

[12] Grechanik, M., Chen Fu, Qing Xie, McMillan, C., Poshyvanyk, D., Cumby, C.: A search engine for finding highly relevant applications. 32nd International Conference on Software Engineering (2010)
[13] Horowitz, B.: A fall sweep. Google Blog, http://googleblog.blogspot.com/2011/10/fall-sweep.html(2011), last retrieved Dec. 2011
[14] Hatcher, E., Gospodnetic, O., McCandless, M.: Lucene in Action (2nd edition). Manning (2010)
[15] Hummel, O., Atkinson, C.: Extreme Harvesting: Test Driven Discovery and Reuse of Software Components. Proceedings of the International Conference on Information Reuse and Integration (2004)
[16] Hummel, O., Atkinson, C.: Using the Web as a Reuse Repository. Proceedings of the International Conference on Software Reuse (2006)
[17] Hummel, O., Janjic, W., Atkinson, C.: Evaluating the efficiency of retrieval methods for component repositories. Proceedings of the International Conference on Software Engineering and Knowledge Engineering (2007)
[18] Hummel, O.: Semantic component retrieval in software engineering. PhD dissertation, University of Mannheim (2008)
[19] Hummel, O., Janjic, W., Atkinson, C.: Code conjurer: Pulling reusable software out of thin air. IEEE Software, Vol.25, No. 5 (2008)
[20] Hummel, O.: Facilitating the comparison of software retrieval systems through a reference reuse collection. Proceedings of the ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation (2010)
[21] Hummel, O., Atkinson, C.: Automated Creation and Assessment of Component Adapters with Test Cases. Symposium on Component-Based Software Engineering (2010)
[22] 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 Engineering, Vol. 31, No. 3 (2005)
[23] Jansen, B.J., Spink, A., Saracevic, T.: Real life, real users, and real needs: a study and analysis of user queries on the web. Information Processing and Management, Vol. 36, No. 2 (2000)
[24] Krueger, C.W.: Software reuse. ACM Computing Surveys, vol. 24, no 2. (1992)
[25] McIlroy, D.: Mass-Produced Software Components. Software Engineering: Report of a conference sponsored by the NATO Science Committee (1968).
[26] Mili, A., Mili, R., Mittermeir, R.: A Survey of Software Reuse Libraries. Annals of Software Engineering 5 (1998)
[27] Mili, A., Yacoub, S., Addy, E., Mili, H.: Toward an engineering discipline of software reuse. IEEE Software, vol. 16, no. 5 (1999)
[28] Nezhad, H., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions. Proceedings of the 16th International Conference on World Wide Web (2007)

[29] Page, L., Brin, S., Motwani, R., Winograd, T.: The Pagerank Algorithm: Bringing Order to the Web. Proceedings of the International Conference on the World Wide Web (1998)

[30] Podgurski, A., Pierce, L.: Retrieving reusable software by sampling behavior. ACM Transactions on Software Engineering and Methodology, Vol.2, No. 3 (1993)

[31] Poulin, J.: Reuse: Been there. Done that. Communications of the ACM. Vol. 42, Iss. 5 (1999)

[32] Prieto-Diaz, R., Freeman, P.: Classifying Software for Reusability. IEEE Software, Vol. 4, No. 1 (1987)

[33] Reiss, S.P.: Semantics-based code search. Proceedings of the 31st International Conference on Software Engineering (2009)

[34] Sahavechaphan, N., Claypool, K.T.: X Snippet: Mining for Sample Code. OOPSLA (2006)

[35] Seacord, R.C.: Software Engineering Component Repositories. Proceedings of the International Workshop on Component-Based Software Engineering (1999)

[36] Seacord, R.C., Hissam, S.A., Wallnau, K.C.: AGORA: a search engine for software components. IEEE Internet Computing, Vol. 2, No. 6 (1998)

[37] Bajracharya, S., Ossher, J., Lopes, C.: Sourcerer: An internet-scale software repository. Proceedings of the ICSE Workshop on Search-Driven Development: Users, Infrastructure, Tools and Evaluation (2009)

[38] Thummalapenta, S. Xie, T.: Parseweb: a programmer assistant for reusing open source code on the web. Proceedings of the International Conference on Automated Software Engineering (2007)

[39] Lemos, O., Bajracharya, S., Ossher, J.: CodeGenie: a tool for test-driven source code search. Proceedings of the International Conference on Object-Oriented Programming (2007)

[40] Szyperski, C.: Component Software: Beyond Object-Oriented Programming (2nd ed.), Addison-Wesley (2002)

[41] Ye, Y. and Fischer, G.: Supporting reuse by delivering task-relevant and personalized information. Proceedings of the International Conference on Software Engineering (2002)

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

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