Recommending Source Code Examples via API Call Usages and Documentation

Collin McMillan
Department of Computer Science
College of William & Mary
Williamsburg, VA 23185
cmc@cs.wm.edu

Denys Poshyvanyk
Department of Computer Science
College of William & Mary
Williamsburg, VA 23185
denys@cs.wm.edu

Mark Grechanik
Accenture Technology Labs
University of Illinois, Chicago
Chicago, IL 60601
drmark@uic.edu

ABSTRACT

Online source code repositories contain software projects that already implement certain requirements that developers must fulfill. Programmers can reuse code from these existing projects if they can find relevant code without significant effort. We propose a new method to recommend source code examples to developers by querying against Application Programming Interface (API) calls and their documentations that are fused with structural information about the code. We conducted an empirical evaluation that suggests that our approach is lightweight and accurate.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software – *reusable libraries, reuse models.*

1. INTRODUCTION

Online source code repositories contain software projects that already implement certain requirements that developers must fulfill. This code can be reused if found; however, manually finding relevant code among libraries of legacy projects or the jungle of open-source repositories is a daunting task. A typical solution is to match words from user queries to the corpus of source code by analyzing its identifier names, comments, or other textual components. Unfortunately, there is no guarantee that needed functionality will be found because the vocabulary used for identifiers and other parts of source code may not overlap with that of users who submit queries. We believe that usage documentation is more likely to contain terms that are also found in queries since both are intended to describe the desired functionality of the code that users search for. Therefore, we use it in our approach to match keywords in queries to documents describing Application Programming Interface (API) calls.

Programmers routinely use API calls as functional abstractions. Popular libraries like file I/O, network, and GUI are generally used in many different projects. For example, the Java Development Kit (JDK) contains over 1,079 classes exporting over 26,906 API calls that address various aspects of software development.

API call documentation is an attractive source of information for programmers to determine how to use these API calls. Contained in these documentation pages are abundant descriptions of these API calls, their behavior, parameters, and output data. In other words, there are two types of information: 1) *structural* from the location of the API calls in the source code classes (e.g., what classes make what calls), and 2) *textual* from the standard documentation descriptions of how the API calls work and interact. The JDK provides a way to produce large, uniform documentation (e.g., Javadocs[1]) of this kind.

Our work leverages information from the source code's list of API calls and those calls' usage documentation. We use text-based *Information Retrieval* (IR) [2] to associate user queries with documentation of API calls. To recommend API calls, we match these associations with software that uses these calls.

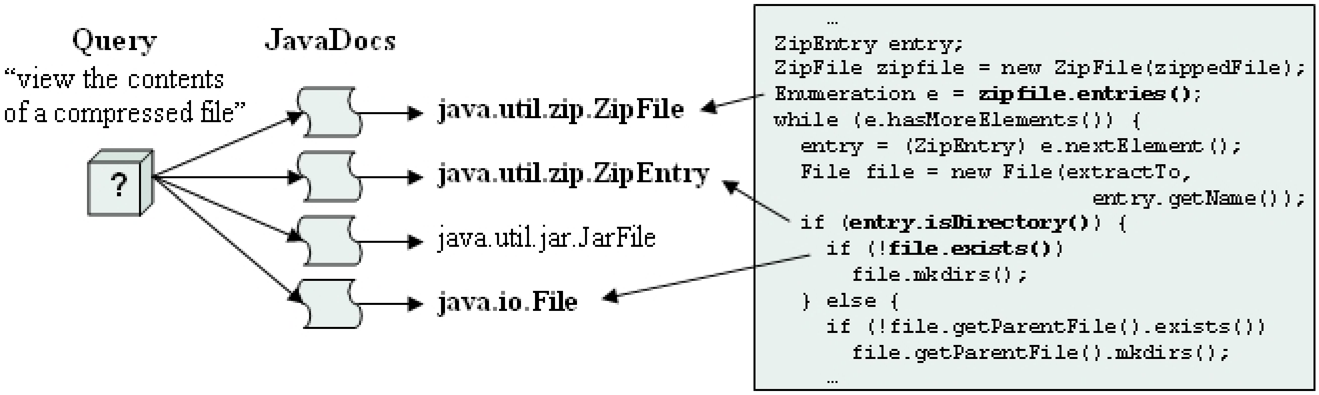
Our approach has the following benefits. First, our approach requires minimal effort on the programmer's part; developers simply enter queries and review retrieved data. Second, we index documentation for textual analysis rather than source code; we believe this may be an advantage because documentation may be more likely to use the same vocabulary as user queries. Third, the links between components of the API (e.g., classes or methods) and their documentation are already defined by vendors and programmers, minimizing the ambiguity in this part of our system. Finally, our empirical evaluation suggests that, when our approach makes recommendations, the correct answer is within the top three results.

2. APPROACH

Figure 1 illustrates our approach with an example. The code snippet shown uses components of the official Java API to extract the contents of an archive in the ZIP file format. In the situation depicted in Figure 1, a programmer wants to view the contents of a compressed archive file, and enters a query to that effect. Our recommendation system connects queries to code by matching them to API usage documentation for calls made within a set of Java example classes. Our implementation is divided into four

RSSE'10, May 2-8, 2010, Cape Town, South Africa.
Copyright 2010 ACM 978-1-60558-974-9/10/05.

[1] http://java.sun.com/j2se/1.5.0/docs/api/



```
  …
ZipEntry entry;
ZipFile zipfile = new ZipFile(zippedFile);
Enumeration e = zipfile.entries();
while (e.hasMoreElements()) {
  entry = (ZipEntry) e.nextElement();
  File file = new File(extractTo,
                        entry.getName());
  if (entry.isDirectory()) {
    if (!file.exists())
      file.mkdirs();
  } else {
    if (!file.getParentFile().exists())
      file.getParentFile().mkdirs();
      …
```

Figure 1. An overview of our approach.

following components.

Documentation Extractor. Classes in the Java API are each associated with a usage document in the JavaDocs. The Documentation Extractor creates a textual corpus from the JavaDocs against which the query can be compared. This corpus is represented in Figure 1 as a mapping of documentation to the Java classes they describe

Bytecode Extractor. Our approach matches queries to Java classes using a list of API calls those classes make. One solution to finding these lists is to extract method calls from the source code for each Java class. Analysis of the source code would introduce a type resolution problem, in which located method calls must be linked to the classes where they are defined. Fortunately, these lists can be readily obtained from the bytecode because the type resolution information is already included at compile-time by the Java compiler. We extract the lists of calls made for every Java class by examining the classes' bytecode with the third-party tool jclassinfo[2]. The bytecode extractor then filters the lists such that they contain only calls to methods for which documentation exists. This gives our approach the advantage of definitively matching API calls to source code, rather than matching based on the source's textual content alone. For example, in Figure 1 the API call *entry.isDirectory()* from the code is connected to the API class *java.util.ZipEntry* – there would be no way to know that *isDirectory()* is a component of *ZipEntry* without structural analysis.

Table 1. Sample listing of software set class, associated description/query, and API calls made

Query, Class	API Calls

"Retrieving the Metadata of Script Engines", MetadataDemo	java.lang.StringBuilder.toString()
	java.lang.StringBuilder.append()
	javax.script.ScriptEngineManager.
	getEngineFactories()
	java.io.PrintStream.println()
	java.util.Iterator.hasNext()
	java.util.Iterator.next()
	javax.script.ScriptEngineFactory.
	getEngineName()
	getEngineVersion()
	getLanguageVersion()
	java.util.List.iterator()

"Using thread to run JavaScript by Java", InterfaceTest	javax.script.ScriptEngineManager.
	getEngineByName()
	java.lang.Thread.join()
	java.lang.Thread.start()
	javax.script.Invocable.getInterface()
	javax.script.ScriptEngine.eval()

Ranking Engine. We use *Latent Semantic Indexing* (LSI) [2] to associate user queries with documentation of API calls. Specifically, with LSI we discover which API classes are relevant to user queries of software functionality. The Ranking Engine ultimately outputs a complete list of Java API classes ranked by the textual similarities of each class's usage documentation.

Source and Documentation Linking Unit. The Bytecode Extractor provides a list of API calls made by each source code example. We link those examples to the list of relevant API calls in six steps:

1. We filter the list of documents provided by the Ranking Engine by keeping only the documents which have a similarity value above a given threshold. The result is the set of relevant API classes, ranked by their similarity to the query. Figure 1 lists four classes from the API that comprise the relevant set.

2. The *method set* of an API class is the set of methods contained by the class. In Figure 1, *entries()* is a member of the method set for *java.util.zip.ZipFile*.

3. We define a *call set* for every example we want to index as the list of API calls that the Bytecode Extractor determines the example makes. The call set for the example in Figure 1 is *java.util.zip.ZipFile.entries()*, *java.util.zip.ZipEntry.isDirectory()*, and *java.io.File.exists()*.

4. We compute a weight for each software example by adding together the number of API calls it makes that are in the list of relevant API classes. More formally:

$$weight = |\{method\ set\} \cap \{call\ set\}|$$

5. Some examples have a large number of calls of one API class, while others make fewer API calls spread across several relevant API classes. We prioritize the latter by adding a 50% bonus to every software class that makes API calls belonging to more than one API class (the idea of this bonus is further explained with an example in Section 4.2):

$$weight = weight * 1.50$$

[2] http://jclassinfo.sourceforge.net/

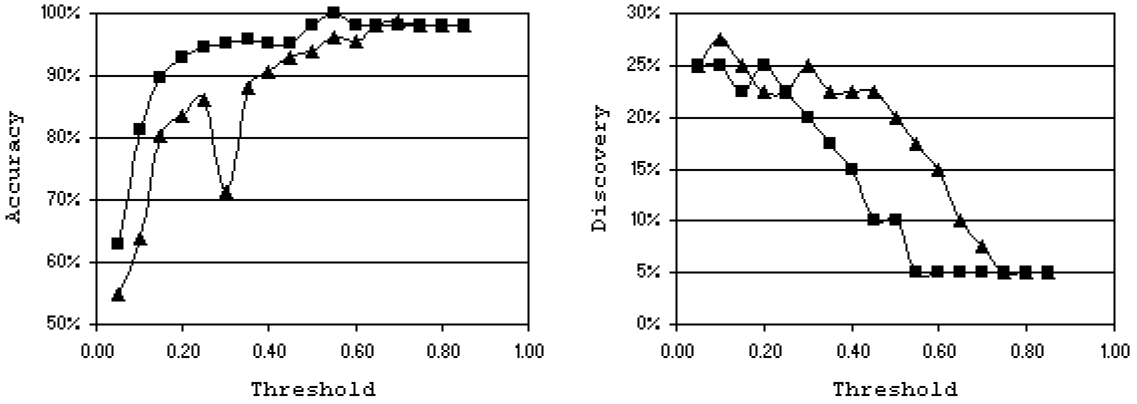


Figure 2: Accuracy and discovery for our approach against the known list.

6. Return the list of classes, organized by their weights in the descending order.

3. EMPIRICAL EVALUATION

We focused on the official Sun Java 1.6 API, a body containing 1,079 classes and a total 26,906 methods. The documentation extractor generated a corpus of 1,079 documents (one for each class in the API) and 31,144 terms. As a searchable software base, we chose a list of 40 publicly available Java examples[3]. The descriptions used to help users locate and organize the examples provide us with sample queries. In this way, the mapping is clear and the software is by design using features provided by the Java API.

Table 1 shows a sample listing of our software base. The API calls column lists the calls made by the class named in the same row. The query is the text description which we use for our analysis – our goal is to link this query to API class documentation for the classes to which the API calls belong.

The list of relevant software classes has a one-to-one mapping with our list of user queries. Unfortunately, the traditional metrics for IR-based search mechanisms, precision and recall, are not highly useful in this case because recall would be either zero or 100% (the single answer is either found or not). Additionally, because our tool's recommendations are returned for inspection by the programmer, the rank of the correct result is also important since we expect that the class in the first position of the suggested results will be examined first. In light of these considerations, we define accuracy and discovery measures as analogs to precision and recall:

$$accuracy = 104 - n * (rank\ of\ correct\ result)$$

$$discovery = \frac{(queries\ with\ positive\ accuracy)}{(total\ number\ of\ queries)}$$

In this way, accuracy measures how well the correct result is ranked, while discovery measures the number of queries that returned any correct recommendation at all. We reason that a programmer may examine at most 25 recommendations. Homan et al. also used only 25 recommendations [5]. Therefore, we chose four for the value of *n* so that accuracy will be positive down to this point. Accuracy will be negative if the correct answer is recommended beyond this point. A correct result in the 5th position will give 80% accuracy, 0% accuracy in the 25th position, etc. Also, the discovery will improve if more queries obtain answers in the top 25.

3.1 Analysis of the Results

We address the following research questions (RQs) through our analysis of the results:

RQ1: What effect does the LSI similarity threshold have on accuracy and discovery when choosing API classes? Figure 2 shows two graphs, one each for accuracy and discovery against a range of thresholds. We ran every query through our system on the corpus described above across the range of thresholds 0.05 to 0.85 in increments of 0.05. We then computed the average accuracy, for every time the correct result occurred in the top 25.

Discovery represents how many times the correct result appears by the 25th suggestion because accuracy is 0% if the correct result is in the 25th position. There is a predictable trade-off of discovery for accuracy as the criterion for similarity becomes more stringent (that is, the threshold is increased). Accuracy is only 62% at the low threshold of 0.10 while discovery is at its peak of 27.5%. The accuracy is conversely at 100% at a threshold of 0.55 while discovery stands at 17.5%. In the first case, the low threshold caused our system to select a large number of API classes. Many of these API classes were not actually relevant, reducing the correct answer's position and lowering accuracy.

RQ2: How many software class recommendations should we return to the programmer? In our evaluation, we assumed that a programmer will not look past the first 25 software classes, but it may be the case that the correct result generally appears at a higher point. Therefore, we want to determine how many results are actually useful to return in order to reduce the programmer's workload. We observe an average accuracy of over 90% at similarity thresholds above 0.15; the correlated discovery is 22.5%. These results mean that, on average, the correct result is supplied within the top three suggestions 22.5% of the time, or not at all. A similar conclusion can be drawn at a threshold of 0.40. These are promising results; with only three choices given, there is minimal clutter for the programmer even when we cannot

[3] http://www.java2s.com/

Table 2. Queries and top five API classes

Query	Weight: API Class
"Retrieving the Metadata of Script Engines"	0.745: java.awt.font.TextAttribute 0.740: java.awt.font.FontRenderContext 0.737: java.awt.Graphics2D 0.665: java.awt.font.GraphicAttribute 0.636: java.awt.font.LineBreakMeasurer
"Using thread to run JavaScript by Java"	0.761: java.util.TimerTask 0.750: java.util.Timer 0.750: java.util.concurrent.Executor 0.743: java.util.concurrent.ThreadFactory 0.726: java.util.concurrent.ThreadPoolExecutor

recommend the correct example.

3.2 Example

To illustrate our approach, consider the user query "*using thread to run JavaScript by Java*" from a user who needs to run JavaScript inside a newly created Java thread. Table 1 shows the API calls that our source code extractor found that it makes from our analyzed Java classes. Table 2 lists the top 5 most relevant API classes to each sample query as ranked by their computed similarity values. For this example, we cropped the list with threshold of 0.40; the entire list of similarities obtained between the query and each API class is too long to list here, but the source and documentation linking unit found that four of these calls matched calls contained by related API classes. Of these, at least two were made to different classes, qualifying the correct answer for a combination bonus of two (50% of the number of unique calls made), resulting in a weight of six. This placed it in second place across the corpus, as shown in Table 3. Therefore, this query was found with 96% accuracy.

LSI did not always report that the queries matched any API documentation, meaning that our system had a peak discovery of only 27.5%. The majority of queries had no relevant classes found. For example, the query "retrieving the Metadata of Script Engines" had no results; in this instance, the ranking engine returned as relevant API classes which had little to do with the required task. The API classes organized under java.awt.font, for example, were found to have textually similar documentation, while in reality providing very different functionality.

4. THREATS TO VALIDITY

We attempted to limit internal threats of validity by using as queries text descriptions of Java source code examples, neither of which were provided by the authors. To confirm our results we need to do the same for multiple datasets as well as conduct a user study of actual queries linked to software classes we know users want to find. Additionally, we need a way to handle multiple correct results, rather than just one. We define our own testing metrics which are untested elsewhere.

One problem in generalizing our results may be the use of only one programming language, though we chose Java, a popular language used in thousands of projects worldwide. Other external threats to validity include the small size of the Java examples comprising our corpus and some imprecision associated with the text parser. Our text parser occasionally makes mistakes when removing HTML; for example, "file" and "Zip" might be in two separate columns of a table, but without the HTML formatting they will appear as one word: "fileZip." Addressing these two challenges is an area of future work.

Table 3. Final results for threshold 0.40

Query	Weight: Relevant Class
"Retrieving the Metadata of Script Engines"	(none found)
"Using thread to run JavaScript by Java"	6: java2s/BlockingDequeTester.class **6: java2s/InterfaceTest.class** 3: java2s/HttpServerDemo.class 3: java2s/Producer.class 3: java2s/Consumer.class

5. RELATED WORK

This work approaches the problem of software reuse in a novel way, but is founded upon widely-accepted techniques and tools. Among these techniques is Latent Semantic Indexing (LSI) [2], a natural language processing technique based on Singular Value Decomposition (SVD) that is used to determine textual similarities among words and documents in large passages of text. For example, LSI can determine how similar the text of a section of documentation is to the text of a method in source code. LSI has already been used in software engineering for a variety of tasks such as concept location [13, 14], impact analysis [16], software reuse [9, 20], identification of abstract data types [10], detection of high level concept clones [11], identification of topics in source code [6], cohesion [12] and coupling [15] measurement

LSI itself works by first creating a term-by-document matrix where every unique word is parsed from a corpus of documents and aligned such that the elements of the matrix correspond to the number of times a given term appears in a given document (a document's term frequency). Queries to the corpus are in the same form (a single-column term-by-document matrix where the query is treated as the document). Unfortunately, unique terms may not have a unique definition as they may be synonyms. Therefore, two related documents may not be considered as such if they use a different vocabulary. LSI overcomes this problem by decomposing the matrix and reducing it into k-dimensional space, where k is the dimensionality reduction factor in SVD [7].

Our approach also integrates structural analysis techniques for looking at source code. A rich community for source examination exists. *Jclassinfo* is a tool for structural analysis of Java byte code providing dependency information about Java classes. It lists every method call made in a class, including ones to the Java API. Other work in the same area includes Sun's own *JavaCompiler* class in the official Java API which offers access to a program's abstract syntax tree while it compiles Java source code. Holmes et al. developed Strathcona [4], an automated mechanism to return relevant source code examples based on nearby structural information and API usage. Our approach expands on previous work by finding relevant API calls from their documentation

We aim at providing complete examples of source code implementing tasks described in user queries. This is similar to research by Little et al. on *keyword programming* [8], except at a higher focus. In Little et al.'s work, the programmer types keywords describing specific methods or actions which are then seamlessly converted into a few lines of source code using relevant API calls. Other tools for locating relevant code from software databases based on user queries exists. For example, PARSEWeb [18] filters results from an existing code search engine by comparing the query to static call information. SpotWeb [19] pinpoints code that is frequently reused. Google Code Search conducts a text-based search over specimens of multiple programming languages, allowing the user refine the search with query parameters. Sourcerer [1] supports both text-based and structural queries through code fingerprints. CodeBroker [21] extracts queries from development activity and returns context-relevant code components. Find-Concept [17] uses natural language processing to locate the source code programmers are looking for during maintenance.

Considering the user query as a code concept or feature to be found, this work is also related to feature location which requires specific parts of documentation to be matched exactly to their implementation in code. A number of applications have been developed for these techniques [3, 14, 17, 22]. Zhao et al. [22] combined knowledge from pseudo-execution traces with candidate links given by the Information Retrieval (IR) methods using a branch-reserving call graph to create a static, non-interactive approach. Eaddy et al. [3] also create a hybrid system, combining information retrieval, execution tracing, and prune dependency analysis (PDA). During execution tracing, activated parts of software are logged while certain known features are activated. Approaches using execution tracing are not applicable in our situation because we are matching natural language text queries to software elements and cannot know beforehand what these queries.

There is published work to index the Java API for user queries. Homan et al. [5] extracted identifiers from Java source code which they manually matched to Java API classes; text similarities of the identifiers to user queries for related API documentation elements were then computed. We instead automatically match API calls to Java classes using structural analysis. Our latest work in the area is Exemplar[4], a source code search engine based on API documentation and usage in Java applications. Exemplar also returns the entire context for relevant software components (e.g., the application and certain included methods), whereas we focus on recommending specific Java classes. Finally, this work uses LSI rather than VSM for textual comparisons and relies on analysis of byte code.

6. CONCLUSIONS

We created an approach to recommend software elements relevant to programmer queries. We link these queries to API usage documentation, that documentation to its described API calls, and then those calls to software classes which use them. We define two metrics for analyzing our results and find during our empirical evaluation that our approach recommends with high accuracy but relatively low discovery – that is, if our system provides the correct result, it will occur within the top three answers. This work is a step towards building effective recommender systems for software reuse by combining source code and usage documentation with both novel and established tools and techniques.

7. ACKNOWLEDGEMENTS

We gratefully acknowledge Chen Fu and Qing Xie for their contributions to this and ongoing work. This work is supported by NSF CCF-0916139, NSF CCF-0916260 and United States AFOSR grant number FA9550-07-1-0030. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

8. REFERENCES

[1] P. Baldi, E. Linstead, C. Lopes, and S. Bajracharya, "A Theory of Aspects as Latent Topics," in *OOPSLA'08*, pp. 543-562.

[2] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by Latent Semantic Analysis," *JASIST,* vol. 41, pp. 391-407, 1990.

[3] M. Eaddy, A. V. Aho, G. Antoniol, and Y. G. Guéhéneuc, "CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis," in *ICPC'08*.

[4] R. Holmes, R. J. Walker, and G. C. Murphy, "Approximate Structural Context Matching: An Approach to Recommend Relevant Examples," *IEEE TSE,* vol. 32, pp. 952-970, Dec. 2006.

[5] M. Homan, A. Robert, and T. Ewan, "Indexing the Java API Using Source Code," in *Proceedings of the 19th ASWEC*'08.

[6] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic Clustering: Identifying Topics in Source Code," *Information and Software Technology,* vol. 49, pp. 230-243, March 2007.

[7] T. K. Landauer and S. T. Dumais, "A Solution to Plato's Problem: The Latent Semantic Analysis Theory of the Acquisition, Induction, and Representation of Knowledge," *Psychological Review,* vol. 104, pp. 211-240, 1997.

[8] G. Little and R. C. Miller, "Keyword programming in java," in *ASE'07*, pp. 84-93.

[9] Y. S. Maarek, D. M. Berry, and G. E. Kaiser, "An Information Retrieval Approach for Automatically Constructing Software Libraries," *IEEE TSE,* vol. 17, pp. 800-813, 1991.

[10]J. I. Maletic and A. Marcus, "Supporting Program Comprehension Using Semantic and Structural Information," in *ICSE'01*, pp. 103-112.

[11]A. Marcus and J. I. Maletic, "Identification of High-Level Concept Clones in Source Code," in *ASE'01*, pp. 107-114.

[12]A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the Conceptual Cohesion of Classes for Fault Prediction in Object Oriented Systems," *IEEE TSE,* vol. 34, pp. 287-300, 2008.

[13]A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," in *WCRE'04*.

[14]D. Poshyvanyk, Y. G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval," *IEEE TSE,* vol. 33.

[15]D. Poshyvanyk and A. Marcus, "The Conceptual Coupling Metrics for Object-Oriented Systems," in *ICSM'06*, pp. 469 - 478.

[16]D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using Information Retrieval based Coupling Measures for Impact Analysis," *Empirical Software Engineering,* 2009.

[17]D. Shepherd, Z. Fry, E. Gibson, L. Pollock, and K. Vijay-Shanker, "Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns," in *AOSD'07*, pp. 212-224.

[18]S. Thummalapenta and T. Xie, "Parseweb: a Programmer Assistant for Reusing Open Source Code on the Web," in *ASE '07*, pp. 204-213.

[19]S. Thummalapenta and T. Xie, "SpotWeb: Detecting Framework Hotspots and Coldspots via Mining Open Source Code on the Web," in *ASE'08*.

[20]Y. Ye and G. Fischer, "Reuse-Conducive Development Environments," *Journal ASE,* vol. 12, pp. 199-235, 2005.

[21]Y. Ye and G. Fischer, "Supporting Reuse by Delivering Task-Relevant and Personalized Information," in *ICSE'02*, pp. 513-523.

[22]W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "SNIAFL: Towards a Static Non-interactive Approach to Feature Location," *ACM TOSEM,* vol.

[4] http://www.xemplar.org/