Improving Software Quality via Code Searching and Mining

Madhuri R. Marri
Department of Computer Science
North Carolina State University
mrmarri@ncsu.edu

Suresh Thummalapenta
Department of Computer Science
North Carolina State University
sthumma@ncsu.edu

Tao Xie
Department of Computer Science
North Carolina State University
xie@csc.ncsu.edu

Abstract

Enormous amount of open source code is available on the Internet and various code search engines (CSE) are available to serve as a means for searching in open source code. However, usage of CSEs is often limited to simple tasks such as searching for relevant code examples. In this paper, we present a generic life-cycle model that can be used to improve software quality by exploiting CSEs. We present three example software development tasks that can be assisted by our life-cycle model and show how these three tasks can contribute to improve the software quality. We also show the application of our life-cycle model with a preliminary evaluation.

1. Introduction

Open source code available on the Internet has become a common platform for sharing source code. Programmers often reuse the design of code examples or adapt code examples of existing open source projects rather than discovering usage patterns by digging into documents. Currently, the amount of open source code available on the Internet is enormous. For example, *sourceforge.net*[1], the world's most popular website for open source software development, hosts about 179,518 projects with two million registered users and a large number of anonymous users. With such enormous amount of open source code available on the Internet, several code search engines (CSE) such as Google code search [6], Krugle [7], Koders [1], Sourcerer [9], and Codase [5] are developed to efficiently search for relevant code examples (i.e., source files containing a search term). These CSEs accept queries such as the names of classes or methods of Application Programming Interfaces (API) and search in CVS or SVN repositories of available open source projects.

Although CSEs can serve as a means for searching in enormous amount of open source code, the usage of CSEs is often limited to simple tasks such as searching for relevant code examples. In this paper, we propose a life-cycle model[2] that combines code searching through CSEs and mining common patterns of API usages from gathered code examples. Our proposed model can be used to assist three main software development tasks: (1) to learn about an API usage by automatically inferring programming rules (from the mined patterns), (2) to use mined patterns to detect defects in a program under analysis, and (3) to infer a fix that needs to be applied for a detected defect.

There exist approaches [4, 8] that mine common usage patterns (e.g., frequent occurrences of pairs or sequences of API method calls) as programming rules for software verification or software reuse. One common characteristic in these existing approaches is that these approaches mine patterns from a few code bases. Therefore, these existing approaches often cannot surface out many programming rules as common patterns because there are often too few data points in these code bases to support the mining of desirable patterns [10]. In other words, the number of data points to support a pattern related to a particular programming rule is often insufficient. The drawback of these approaches is reflected in empirical results reported by these existing approaches: often a relatively small number of real programming rules were inferred from huge code bases.

A natural question to ask is whether a larger number of code bases (such as a large scale of open source code) can serve as an alternative data source for smaller code bases. One issue with a larger number of code bases is that mining a larger number of code bases is often not scalable. To address this issue, we propose a life-cycle model based on code searching and mining. In our life-cycle model, we expand the data scope to a larger number of code bases and include techniques to address scalability issues. In particular, we search for relevant code examples using CSEs and mine *only* those code examples. Our life-cycle model can assist in improving software quality over different phases of software development. We refer to our model as life-cycle model since the mined patterns can be used for writing new code, which can again be used as input for our model.

[1] `http://sourceforge.net/`

[2] The term life-cycle model is inspired from software development life cycle and refers to the life cycle of mining patterns.

We applied our life-cycle model in our previous approach, called PARSEWeb [10], that identifies frequent method-invocation sequences to serve as solutions for queries of the form "*Source object type* → *Destination object type*". In the evaluation of PARSEWeb, we show that our model can address issues that cannot be addressed by a CSE or any existing mining approach individually. We also show that code examples gathered from a CSE require post-processing before mining common usage patterns. In this paper, we elaborate on the life-cycle model of code searching and mining. We describe issues (and related post-processing techniques) that need to be addressed before using gathered code examples for mining patterns. We also present the application of our model in improving software quality with an emphasis on the post-processing techniques.

2. Life-Cycle Model

We next describe our life-cycle model that exploits CSEs and mines common patterns from gathered code examples. These common patterns can be used in improving software quality. Our model includes two phases: *searching* and *mining*. In the searching phase, we use CSEs to gather relevant code examples. In the mining phase, we analyze these code examples and mine patterns that describe how to use an API. Figure 1 presents an overview of our life-cycle model. We next describe each phase in detail.

2.1 Searching

The searching phase includes two tasks: *query construction* and *duplicate elimination*.

Query Construction. In the query construction task, we construct queries with API names as search terms. For example, we construct the query "`lang:java org.apache.regexp.RE`" to gather relevant code examples of the `RE` class from Google code search (GCS). These code examples show how to use the `RE` class provided by the Apache library [3]. GCS returns around $2,000$ code examples for this query. Based on our experience with CSEs, one observation with query construction is that the *relevance*[3] of resulting code examples mainly depends on the format of the query issued to CSEs. Without a well-formulated query, CSEs can result in a high number of irrelevant code examples. For example, to search for relevant code examples of the `fopen` API, a basic search query on GCS is "`lang:c fopen`". This query returns around $752,000$ code samples. When the query is tuned to "`lang:c file:.c$ [\ s \ *]fopen [\ s]?\(`" (GCS supports search with regular expressions), GCS returns $689,000$ code examples. Among the top 50 returned code examples, the number of relevant code examples was found to be doubled among code examples returned by a specific query (query with regular expressions) when compared to that of a basic query. The

[3] A code example is relevant when it includes a call site of the required API that is searched for.

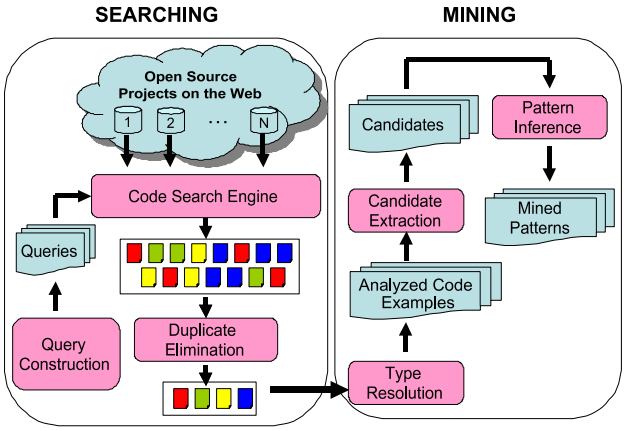

Figure 1. Phases in the life cycle of mining approaches based on code search engines

relevance (or quality) of gathered code examples plays an important role in mining common patterns from the gathered code examples. Although a programmer can decide to filter out irrelevant code examples during either the searching or mining phase, filtering out irrelevant code examples using an appropriate query in the searching phase can help reduce additional efforts in handling irrelevant code examples.

Duplicate Elimination. One observation with code examples returned by CSEs is that these code examples often include duplicate copies. We consider two code examples as duplicate of each other, if both belong to the same project and the same source file. For example, among the $2,000$ code examples returned by GCS for the query "`lang:java org.apache.regexp.RE`", the source file `JakartaRegexpRegexp.java` is found 13 times. Among these 13 copies, there are 5 different versions of the source file and the remaining 8 copies are duplicates of these 5 versions. There are both desirable and undesirable consequences with duplicate or multiple versions of source files among code examples. For example, code examples that are duplicate of the same source file, such as those belonging to a particular jar file, can be found to be used in various projects. The existence of duplicate or multiple copies for a code example can indicate that the code example is widely used and therefore the code example can be trusted more than those code examples that do not have duplicate or multiple versions. On the other hand, duplicate or multiple copies can bias the results of mining approaches that try to mine common patterns. To mine unbiased patterns used across a large number of code bases, we propose duplicate elimination to identify and filter out duplicate code examples.

2.2 Mining

The mining phase includes three tasks: *type resolution*, *candidate extraction*, and *pattern inference*. We refer to

```
01:import java.util.ArrayList;
02:import java.util.*;
03:Public class test {
04:  public void method1(ArrayList list) {
05:  Iterator iter = list.iterator();
06:  while(iter.hasNext()) {
07:   String str = (String) iter.next();
08:  ...}}
```

Figure 2. A code example using `Iterator` API.

these three tasks as *post-processing techniques* on gathered code examples.

Type Resolution. In the type resolution task, we resolve object types such as the return object type of a method call in gathered code examples. These object types are necessary for analyzing gathered code examples. In our model, we cannot use traditional techniques for parsing and resolving object types. The primary reason is that CSEs often return only individual source files (i.e., code examples) including the search term, and these code examples are often partial and not compilable. In our context, a partial code example indicates that the code example is complete; however, the other source files on which the code example is dependent upon are not available. To achieve the task of type resolution, we use partial program analysis for resolving object types. In our PARSEWeb approach [10], we developed 16 heuristics and these heuristics are contrary to type checking done by a compiler. We next present a sample heuristic for inferring fully qualified names using a code example (shown in Figure 2) to show the use of these heuristics in analyzing partial code examples.

Inferring fully qualified names. In Java, classes and interfaces have fully qualified names that can be extracted from the class declaration. However, as our gathered code examples are partial, we infer fully qualified names from `import` statements in these code examples. For example, the fully qualified name of the `ArrayList` class is inferred from the `import` statement in Line 1. However, this heuristic cannot infer the fully qualified name for the `Iterator` class referred in Line 5. The reason is that the related `import` statement in Line 2 uses `*` instead of the `Iterator` class. Our heuristics are not complete as these heuristics cannot resolve entire type information. However, the evaluation results of our PARSEWeb approach show that these heuristics are often effective in resolving required type information.

Candidate Extraction. In the candidate extraction task, we analyze gathered code examples to extract pattern candidates. These pattern candidates include information about API usage. For example, a pattern candidate extracted from the code example in Figure 2 is "`Iterator.next` should be preceded with a `boolean` check on `Iterator.hasNext`".

Pattern Inference. In the pattern inference task, we apply mining techniques such as frequent subsequence mining [2] on extracted pattern candidates to mine common patterns of API usage. The details of these two tasks (candidate extraction and pattern inference) vary across the types of problems being addressed using our model, whereas type resolution is an essential task to resolve type information in gathered code examples.

3. Application of the Life-Cycle Model

In this section, we describe example software development tasks that can be assisted by our life-cycle model and show the utility of our model with a preliminary evaluation done for one of these tasks.

3.1 Tasks Assisted by Life-Cycle Model

We expect that our life-cycle model can be used to improve software quality over different phases of software development by assisting three example major tasks.

Development. The patterns mined using our life-cycle model can be used to assist programmers during the development phase. These mined patterns provide common usage scenarios of how to reuse APIs and can be referred to as specifications while writing code. We implemented an approach, called PARSEWeb [10], based on our life-cycle model. PARSEWeb can be used to assist programmers during software development. The utility of the PARSEWeb approach over a traditional approach based on a CSE is shown in Section 3.2.

Verification. The patterns mined using our life-cycle model can be used to detect deviant behavior in a program under analysis. These patterns can be treated as specifications of an API usage and any deviation from the pattern in a program under analysis can be reported as a violation. For example, consider a pattern mined for the `fopen` API of standard C library (`stdio.h`) as shown below:

```
API method: fopen
Condition check on "return" value
Condition Type: NULL-CHECK
```

The preceding pattern describes that a majority of gathered code examples contain a `NULL` condition check on the return value of the `fopen` method call. This pattern can be used to detect defects related to missing condition checks after the `fopen` API call in the program under analysis. This example illustrates that our life-cycle model can be used to detect defects in the verification task.

Maintenance. The patterns mined using our life-cycle model can also be used for suggesting defect fixes during the maintenance task. For example, consider the following pattern related to the `Iterator.next` method:

```
API method: Iterator.next
Condition check on "return" value of
Iterator.hasNext
Condition Type: BOOLEAN-CHECK
```

The preceding pattern describes that there should be a `boolean` check on the `Iterator.hasNext` method before invoking the `Iterator.next` method. Failing to perform the boolean check can cause `NoSuchElementException`. Consider that the verification task detects a violation of the preceding pattern in a program under analysis. In this scenario, we can suggest a defect fix based on the pattern. For example, we can automatically perform defect fixing by inserting a `boolean` check on the `Iterator.hasNext` method before the `Iterator.next` method.

3.2 Preliminary Evaluation

We next show the utility of our life-cycle model over a traditional approach of directly searching via a CSE with an example task related to software development. We implemented our life-cycle model in our previous approach, called PARSEWeb [10]. PARSEWeb accepts queries of the form "*Source object type* → *Destination object type*" and finds method-invocation sequences that produce the destination object type from the source object type.

We use a programming problem "`org.eclipse.ui.IWorkbenchWindow` → `org.eclipse.ui.IViewPart`" described in a previous related approach [8]. The programming problem can be interpreted as that a programmer has an object of the `IWorkbenchWindow` class, and the programmer wants a method-invocation sequence to obtain an object of the `IViewPart` class. We use four code search engines (GCS, Koders, Krugle, and Codase) and PARSEWeb to investigate how they can assist in addressing this programming problem.

The minimal requirement for a code example to include a solution method-invocation sequence is that the code example should include both classes `IWorkbenchWindow` and `IViewPart`. Therefore, we constructed a query with both class names as search terms and used all four CSEs to gather relevant code examples. GCS, Krugle, Koders, and Codase returned 775, 112, 478, and 0 code examples, respectively. We inspected the top ten code examples returned by each CSE to check whether these code examples include a solution method-invocation sequence. We found that GCS and Koders include a solution method-invocation sequence in the sixth and eighth code examples, respectively. We could not find any solution method-invocation sequence among the code examples returned by Koders and Codase. There are two major tasks that need to be carried out in using CSEs directly for addressing this programming problem. First, the programmer has to browse to the sixth or eighth code example for getting a solution sequence. Second, the programmer does not have any knowledge whether this solution sequence is a commonly used method-invocation sequence. We next used PARSEWeb to recommend a solution for the programming problem. The solution sequence recommended by PARSEWeb is shown as below.

```
... IWorkbenchWindow iwwObj;
IWorkbenchPage iwpObj = iwwObj.getActivePage();
IViewPart ivpObj = iwpObj.findView(String);
```

PARSEWeb analyzed code examples gathered from GCS to generate sequence candidates. PARSEWeb used these sequence candidates to mine commonly used method-invocation sequence. PARSEWeb recommended a single solution sequence that is a common sequence among gathered code examples. This example shows the utility of our life-cycle model used to develop our PARSEWeb approach.

4. Conclusion

We proposed a life-cycle model that can be used to develop approaches based on code searching and mining. We elaborated on the two phases of our life-cycle model and suggested post-processing techniques for mining patterns from gathered code examples. We also presented three example software development tasks (that contribute to improve software quality) that can be assisted by our life-cycle model. Additionally, we highlighted the utility of our life-cycle model with an approach developed based on our life-cycle model.

Acknowledgments. This work is supported in part by NSF grant CCF- 0725190, ARO grant W911NF-08-1-0443, and ARO grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI).

References

[1] Koder's Zeitgeist. `http://www.koders.com/zeitgeist/`.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. VLDB*, pages 487–499, 1994.

[3] The Apache Jakarta Project, 2007. `http://jakarta.apache.org/regexp/`.

[4] R.-Y. Chang, A. Podgurski, and J. Yang. Finding what's not there: a new approach to revealing neglected conditions in software. In *Proc. ISSTA*, pages 163–173, 2007.

[5] Codease, 2005. `http://www.codase.com/`.

[6] Google Code Search Engine, 2006. `http://www.google.com/codesearch`.

[7] V. Magotra. The art of ranking code search results, 2006. `http://blog.krugle.com/?p=184`.

[8] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proc. PLDI*, pages 48–61, 2005.

[9] B. Sushil, N. Trung, L. Erik, D. Yimeng, R. Paul, B. Pierre, and L. Cristina. Sourcerer: a search engine for open source code supporting structure-based search. In *Proc. OOPSLA Companion*, pages 681–682, 2006.

[10] S. Thummalapenta and T. Xie. PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proc. ASE*, pages 204–213, 2007.