MAPO: Mining API Usages from Open Source Repositories

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
xie@csc.ncsu.edu

Jian Pei
School of Computing Science
Simon Fraser University
Burnaby, BC Canada V5A 1S6
jpei@cs.sfu.ca

ABSTRACT

To improve software productivity, when constructing new software systems, developers often reuse existing class libraries or frameworks by invoking their APIs. Those APIs, however, are often complex and not well documented, posing barriers for developers to use them in new client code. To get familiar with how those APIs are used, developers may search the Web using a general search engine to find relevant documents or code examples. Developers can also use a source code search engine to search open source repositories for source files that use the same APIs. Nevertheless, the number of returned source files is often large. It is difficult for developers to learn API usages from a large number of returned results. In order to help developers understand API usages and write API client code more effectively, we have developed an API usage mining framework and its supporting tool called MAPO (for Mining API usages from Open source repositories). Given a query that describes a method, class, or package for an API, MAPO leverages the existing source code search engines to gather relevant source files and conducts data mining. The mining leads to a short list of frequent API usages for developers to inspect. MAPO currently consists of five components: a code search engine, a source code analyzer, a sequence preprocessor, a frequent sequence miner, and a frequent sequence postprocessor. We have examined the effectiveness of MAPO using a set of various queries. The preliminary results show that the framework is practical for providing informative and succinct API usage patterns.

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

General Terms: Design, Documentation, Measurement.

Keywords: Application Programming Interfaces, Program Comprehension, Mining Software Repositories.

1. INTRODUCTION

During software development, by invoking the corresponding APIs, developers often reuse existing class libraries or frameworks to write client code. These APIs, being equipped with only simple API documents, however, are often complex and not well documented. For example, suppose we plan to use the Byte Code Engineering Library (BCEL) [9] to instrument the bytecode of a Java class by adding an extra method to the class (This programming task was faced by the first author when developing a dynamic analysis tool). By a quick search on BCEL's API document, we can find a class called `org.apache.bcel.generic.ClassGen` containing a method called `public void addMethod(Method m)`, which seems to be the right API method to use. From the API document for this method, we only see a simple description for the method: "*Add a method to this class. Parameters: m - method to add*." We still do not know how to use this method, in particular, how to prepare the `Method` object, what method calls should be invoked on this `Method` object before `addMethod` is invoked, and what method calls are needed to be invoked on the `ClassGen` object before and after the `addMethod` is invoked, and so forth.

Because the API document does not provide sufficient information for us to learn how to use the API, we can search the Web using a general search engine, say Google, to look for other developers' experience of using the API. We can indeed find some articles that include code segments to briefly explain specific usages of BCEL. However, because the same API can be used in different ways, we still do not have high confidence on whether the described code segments represent the API usage that we should follow. We can also use some code search engines such as the Koders search engine [3] and the SPARS-J search engine [5, 13]. These search engines retrieve from open source repositories a long list of source files that contain the call sites of the `addMethod` method. Nevertheless, the numerous and improperly sorted results returned by those source code search engines cannot quickly and comprehensively help us understand the commonality among these source files.

Only collecting a set of call sites or code segments is far from enough to support developers' learning of API usage. Developers are interested in the inherent usage patterns of APIs. Thus, the real challenge is how to construct *a tool to analyze the code segments and disclose the inherent usage patterns*, which motivates this research.

In order to help developers understand API usages and write API client code more effectively, we have developed an API usage mining framework and its supporting tool called MAPO (for Mining API usages from Open source repositories) by leveraging the existing code search engines. The mining produces a short list of frequent API usage patterns for developers to inspect. MAPO consists of five components: a code search engine, a source code analyzer, a sequence preprocessor, a frequent sequence miner, and a frequent sequence postprocessor. To examine its effectiveness, we have applied the MAPO tool on a set of various queries. The preliminary results show that the framework is practical for providing informative and succinct API usage patterns.

2. MAPO DESIGN CONSIDERATIONS

In MAPO, we want to achieve the following four objectives.

1. The tool should be able to extract API usage information from a source file that may not be able to be compiled by a compiler, because a source code search engine may not return all other source files that the source file depends on.
2. The tool should be able to infer frequent API usages that include sequencing information among method calls. The sequencing information is an important part of API usages. For example, the `open` method of a `File` object needs to be invoked before the `read` method.
3. The tool should be able to mine frequent API usages that include method calls from more than one class, because realistic API usages often involve methods from multiple classes.
4. The tool should be able to produce a short list of relevant frequent API usage patterns for inspection.

3. CHOICES OF MINING TOOLS

Given a set of code segments, a user wants to obtain the common usage patterns of APIs. A few data mining techniques may be applicable in such a situation.

Straightforwardly, for each code segment, we can obtain the set of APIs used in the segment. Then, we can mine the combinations of APIs appearing in many segments by applying the frequent itemset mining methods such as Apriori [6] and FP-growth [10]. Given a transaction database where each transaction is a set of items, and a minimum support threshold min_sup, a frequent itemset mining method returns the complete set of item combinations that appear in at least min_sup transactions.

Frequent itemset mining provides the insights on which APIs are frequently used together in code segments. However, it still does not fully disclose the usage patterns. Particularly, frequent itemsets do not indicate how a group of APIs may be invoked in some specific order.

To capture the groups of APIs that are frequently used together as well as the orders in which they are used, we mine sequential patterns [7]. Given a database of sequences and a minimum support threshold min_sup, a sequential pattern mining algorithm returns the complete set of frequent subsequences, called *sequential patterns*, that appear in at least min_sup sequences in the database.

The complete set of sequential patterns are informative for API usage analysis. It, however, may contain redundant information. For example, suppose methods `open`, `read` and `close` of a `File` object are always called in the order of `open-read-close`. Then, ⟨`open`⟩, ⟨`read`⟩, ⟨`close`⟩, ⟨`open`; `read`⟩, ⟨`open`; `close` ⟩, ⟨`read`; `close`⟩, and ⟨`open`; `read`; `close`⟩ are all sequential patterns with the same frequency in the database. Pattern ⟨`open`; `read`; `close`⟩ should be used as the representative of the whole group of sequential patterns because it captures the complete usage information that `open`, `read`, and `close` are used. Once pattern ⟨`open`; `read`; `close`⟩ is identified, the other six patterns in the group become redundant because they are sub-patterns of ⟨`open`; `read`; `close`⟩ and have the same frequency. A pattern S such as ⟨`open`; `read`; `close`⟩ is called a *closed sequential pattern* if it is frequent and there exists no any proper super-pattern of S having the same frequency as S. In the API usage mining task, the complete set of closed sequential patterns gives the complete yet non-redundant information on the common usage patterns of APIs.

Finite state automaton (FSA) learning has been frequently used to learn API protocols in the form of an FSA out of program-execution traces [8, 16]. Given a set of sequences, an FSA learning algorithm reconstructs an FSA that can accept these sequences.

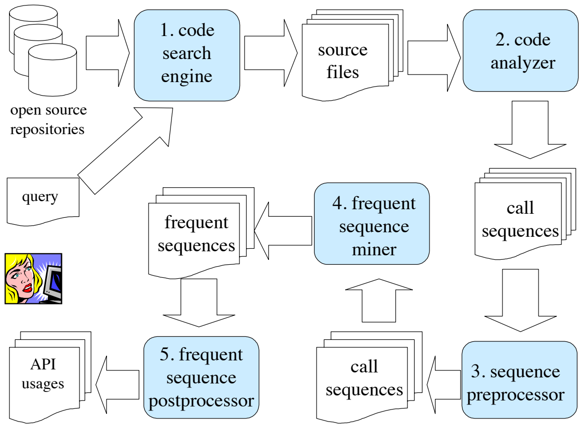

Figure 1: An overview of the API usage mining framework.

In our research context, the sequences extracted from the results of code search engines can include many irrelevant method calls; therefore, applying FSA learning in our research context would produce a large FSA, which is often not useful and not focusing. A probabilistic FSA learning algorithm [18] can be used to infer a probabilistic FSA where each edge is weighted by how often the edge is traversed while accepting sequences. Infrequent behavior reflected by those rarely-traversed edges can be removed to reduce the complexity of the learned FSA. The resulting FSA, however, would still be complicated.

4. API USAGE MINING FRAMEWORK

To automatically mine API usages from open source repositories, we have developed a novel framework based on existing code search engines and a frequent sequence miner. Figure 1 shows the overview of the framework. The framework receives a query describing a method name, class name, or package name, and outputs a set of API usages (in the form of method call sequences). The framework consists of five major components: a code search engine, a code analyzer, a sequence preprocessor, a frequent-sequence miner, and a frequent-sequence postprocessor. The code search engine receives a query and then searches open source repositories for source files that are relevant to the query. The code analyzer analyzes the relevant source files returned by the code search engine and produces a set of method call sequences, each of which is a callee sequence for a method defined in the source files. The sequence preprocessor inlines some call sequences into others based on caller-callee relationships and removes some irrelevant call sequences from the set of call sequences according to the given query. The frequent-sequence miner discovers frequent sequences from the preprocessed sequences. The frequent-sequence postprocessor reduces the set of frequent sequences in some ways. We next illustrate each of the components in the framework in detail.

4.1 Code Search Engine

There exist a number of code search engines[1]. Among the non-academic search engines, we found that Koders [3], CodeBase [1], and DocJar [2] can return a list of Java source files given the textual query of "`bcel`." SPARS-J [5] developed by Inoue et al. [13] is one of the few academic code search engines. Like the preceding non-academic engines, SPARS-J can also return source files given the textual query of "`bcel`." Currently we have not committed our development efforts to develop a tool for automatically grabbing

[1]`http://gonzui.sourceforge.net/links.html`

source files returned by various search engines. Instead, we manually download source files from the returned set of links. We plan to implement a tool to automate this task in the future. Note that although our framework is based on a code search engine, the source files used for API usage mining can also be collected directly from any source repositories, such as local source repositories within a software company or a combination of open and local source repositories.

4.2 Source Code Analyzer

To extract method-call sequences from source files, we have developed a source code analyzer based on a lightweight source code analyzer PMD [4], which does not require source files to be compilable. From each source file, the code analyzer extracts a list of methods, each of which is associated with a sequence of method calls invoked by the method. Currently we count the number of method parameters to characterize method signatures in method-call names. We ignore control flows but simply use call-site locations when extracting method-call sequences. Note that method-call sequences can include method calls from more than one class; therefore, we can later infer from them API usages involving more than one class.

We incorporate various techniques in the source code analyzer to try to collect the class name and the full package name (called full class name) for each method call. The preceding task is not trivial when the classes that a source file depends on are unavailable. We keep track of field declarations and local variable declarations so that we can know the class name of a method call based on the receiver object name. We keep track of `import` statements so that we can know the full package name of a class based on the class name if the class is explicitly exported in the `import` statements. We construct a map from method-call names to their full class names across various source files in the first pass of the analysis. Then, in the second pass, for those method calls whose full class names cannot be found, we assign to them full class names if we can find the same method-call names in the map. In the end, we filter out from the extracted method call sequences those method calls whose full class names cannot be found.

4.3 Sequence Preprocessor

We have developed several techniques to improve the quality of extracted method-call sequences before they are fed to a mining tool. First, we filter out those method calls whose full class names start with "`java.`": those are commonly used Java library classes. Including them in the sequences is often not necessary.

Second, when `ee` is the callee of a caller `er`, and the method-call sequence of `ee` is `ms`, we inline `ee` by replacing all occurrences of `ee` with `ms` in the method-call sequence of `er`. We perform the inlining process for three iterations by default, allowing us to collect method sequences up to the call depth of three. Note that if a call sequence pattern is spread across several source files or the call depth of three, MAPO cannot recognize it completely.

Finally, from the set of inlined sequences, we remove method-call sequences that do not contain the given query entity (e.g., method name, class name, or package name), because these sequences are not relevant to the given query entity.

4.4 Frequent Sequence Miner

We use the BIDE [19] algorithm to mine closed sequential patterns from the preprocessed method-call sequences. BIDE enumerates closed sequential patterns in a depth-first search. For example, suppose A, B, C, and D are the APIs in question. Then, the complete set of closed sequential patterns can be divided into four exclusive subsets: the ones having $\langle A \rangle$, $\langle B \rangle$, $\langle C \rangle$, and $\langle D \rangle$ as prefixes, respectively. Each subset is further divided recursively.

In the depth-first search, once a sequence S is encountered whose frequency in the database is smaller than the support threshold, BIDE does not need to search any longer sequence S' that has S as a prefix, because the frequency of S' cannot exceed that of S.

Moreover, once a frequent sequence S is met, all APIs that appear in every sequence that contains S in the database are also extracted and a closed sequential pattern is formed. On the other hand, if S is a sub-sequence of a closed sequential pattern S' and S and S' have the same frequency, then BIDE does not need to recursively search the subtree of S, because it is not closed.

BIDE also uses a few techniques to speed up the search, such as searching using projected databases and the pseudo-projection technique. Limited by space, we omit the details here.

4.5 Frequent Sequence Postprocessor

Because the number of frequent sequences mined by BIDE could be large, we have developed several techniques to reduce the size of frequent sequences without compromising important API usage information. First, we remove frequent sequences that do not contain the given query entity (e.g., method name, class name, or package name), because these frequent sequences are not relevant to the query entity. Second, in frequent sequences, we compress consecutive calls of the same method into one. Alteratively we can compress method-call sequences in a similar way in the sequence preprocessor but doing compression here can help compress repetitive call patterns separated by infrequent method calls in original call sequences. Third, we remove duplicate frequent sequences after the compression. Finally, we further reduce the set of frequent sequences so that every frequent sequence in the reduced set is not a subsequence of another in the reduced set. We adapted the implementation of the longest common sequences (LCS) [11] algorithm to implement the subsequence checking.

5. PRELIMINARY RESULTS

We have applied MAPO on various queries. This section shows two particular queries that are related to the motivating example shown in Section 1. We searched Koders [3] with two queries: the textual BCEL query of "`org.apache.bcel.generic ClassGen addMethod`" and the textual Javaassist query of "`javassist CtClass addMethod`," where we replace those dots that separate package names, class names, and method names with space characters in order to allow Koders to return more related results. The method in the Javaassist query has the same functionality as the method in the BCEL query but these two methods are from two different libraries.

Table 1: API usage mining results

query	#files	#seqs	#seqs-pre	#freqseq	#freqseq-post
BCEL	36	1087	186	429	8
Javassist	50	828	141	90	23

Table 1 show the statistics of the API usage mining results for these two queries. Columns 1-6 show the query name, the number of source files returned by Koders, the number of sequences extracted by the code analyzer, the number of sequences produced by the sequence preprocessor, the number of frequent sequences produced by BIDE, and the number of frequent sequences produced by the frequent sequence postprocessor, respectively. From the statistics, we can observe that the sequence preprocessor is effective in reducing the size of sequences before being fed to BIDE, and the frequent sequence postprocessor is also effective in reducing the size of frequent sequences before being inspected by users.

We inspected the frequent patterns produced by MAPO and found them precise in general in characterizing the API usages. For example, for the BCEL query, the first frequent sequence is listed as below where package names are omitted for simplicity, the enclosed numbers represent the total number of method parameters if any, and "`<init>`" represents a constructor call.

```
InstructionList.<init>
InstructionFactory,createLoad(2)
InstructionList,append(1)
InstructionFactory,createReturn(1)
InstructionList,append(1)
MethodGen,setMaxStack
MethodGen,setMaxLocals
MethodGen,getMethod
ClassGen,addMethod(1)
InstructionList.dispose
```

The API usages mined by MAPO for the Javassist query are simpler and also more diverse than the ones for the BCEL query. The first frequent sequence is simply two method calls:

```
CtNewMethod,make(2)
CtClass,addMethod(1)
```

6. RELATED WORK

CodeWeb developed by Michail [17] mines association rules such as that application classes inheriting from a particular library class often instantiate another class or one of its descendants. MAPO focuses on API usages in general beyond library reuse patterns through class inheritances. In addition, MAPO mines API usages that include sequencing information among method calls. PR-Miner developed by Li and Zhou [14] uses frequent itemset mining to extract implicit programming rules and detect their violations for detecting bugs. The rules mined by PR-Miner do not include sequencing information, which is mined by MAPO. A tool developed by Williams and Hollingsworth [20] and DynaMine developed by Livshits and Zimmermann [15] mine simple rules from software revision histories. These rules involve mostly method pairs, whereas MAPO mines more complicated API usage patterns from code segments returned by a code search engine. Different from all the preceding mining tools, which take no query before mining, MAPO takes a query and mines from code segments relevant to the query.

MAPO is related to a number of code search engines [3,5,13] as well as the Strathcona tool developed by Holmes and Murphy [12]. Strathcona locates a set of relevant code examples from an example repository by matching the structure of the code under development with the code examples in the repository. Like other code search engines, Strathcona returns a list of relevant code examples, whereas MAPO can extract common patterns among the list of relevant code examples returned by a code search engine or even Strathcona.

7. CONCLUSIONS

In order to help developers understand API usages and write API client code more effectively, we have developed a novel framework and its supporting tool called MAPO for mining API usages from open source repositories by leveraging existing code search engines and a frequent sequence miner. MAPO can produce a short list of succinct frequent sequences for inspection. Our preliminary results show that MAPO provides useful API usage patterns for developers to inspect. In future work, we plan to synthesize code fragments from mined frequent API usages. These synthesized code fragments can be directly inserted into developers' code.

Acknowledgments

We would like to thank Jiawei Han and Jianyong Wang for helping us to use the BIDE tool.

8. REFERENCES

[1] CodeBase, 2005. `http://www.codase.com/`.

[2] DocJar, 2005. `http://www.docjar.com/`.

[3] The Koders source code search engine, 2005. `http://www.koders.com`.

[4] PMD, 2005. `http://pmd.sourceforge.net/`.

[5] SPARS-J, 2005. `http://demo.spars.info/`.

[6] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases*, pages 487–499, Sept. 1994.

[7] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. 1995 Int. Conf. Data Engineering*, pages 3–14, Taipei, Taiwan, Mar. 1995.

[8] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[9] M. Dahm and J. van Zyl. Byte Code Engineering Library, April 2003. `http://jakarta.apache.org/bcel/`.

[10] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data*, pages 1–12, May 2000.

[11] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM*, 24:644–675, 1977.

[12] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proc. 27th International Conference on Software Engineering*, pages 117–125, 2005.

[13] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Ranking significance of software components based on use relations. *IEEE Transactions on Software Engineering*, 31(3):213–225, March 2005.

[14] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. ESEC/FSE*, pages 306–315, 2005.

[15] B. Livshits and T. Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *Proc. ESEC/FSE*, pages 296–305, 2005.

[16] L. Mariani and M. Pezzè. Behavior capture and test: Automated analysis of component integration. In *Proc. 10th International Conference on Engineering of Complex Computer Systems*, pages 292–301, June 2005.

[17] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proc. 22nd International Conference on Software Engineering*, pages 167–176, 2000.

[18] A. V. Raman and J. D. Patrick. The sk-strings method for inferring pfsa. In *Proc. Workshop on Automata Induction, Grammatical Inference and Language Acquisition*, 1997.

[19] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *Proc. 20th International Conference on Data Engineering*, pages 79–90, 2004.

[20] C. C. Williams and J. K. Hollingsworth. Recovering system specific rules from software repositories. In *Proc. 2005 International Workshop on Mining Software Repositories*, pages 1–5, 2005.