Mining Succinct and High-Coverage API Usage Patterns from Source Code

Jue Wang[§*], Yingnong Dang[5], Hongyu Zhang[§*], Kai Chen[€*], Tao Xie[ɜ*], Dongmei Zhang[5]
[§]Tsinghua Univ., China, [5]Microsoft Research Asia, China, [€]Peking Univ. China, [ɜ]NC State University, NC, USA
[§]cecilia.juewang@gmail.com, [§]hongyu@tsinghua.edu.cn, [5]yidang, dongmeiz@microsoft.com, [ɜ]xie@csc.ncsu.edu,
[€]chenkai18@pku.edu.cn

***Abstract*—During software development, a developer often needs to discover specific usage patterns of Application Programming Interface (API) methods. However, these usage patterns are often not well documented. To help developers to get such usage patterns, there are approaches proposed to mine client code of the API methods. However, they lack metrics to measure the quality of the mined usage patterns, and the API usage patterns mined by the existing approaches tend to be many and redundant, posing significant barriers for being practical adoption. To address these issues, in this paper, we propose two quality metrics (succinctness and coverage) for mined usage patterns, and further propose a novel approach called Usage Pattern Miner (UP-Miner) that mines succinct and high-coverage usage patterns of API methods from source code. We have evaluated our approach on a large-scale Microsoft codebase. The results show that our approach is effective and outperforms an existing representative approach MAPO. The user studies conducted with Microsoft developers confirm the usefulness of the proposed approach in practice.**

***Index Terms*—API usage, usage pattern, sequence mining, software reuse, mining software repositories.**

I. Introduction

Application Programming Interface (API) is an important form of software reuse and it has been widely used. It is common for an API method, no matter whether it is public (e.g., .NET framework API methods and Java Development Kit) or private, to be used in different contexts to complete different tasks. An API method combined with other API methods that should be invoked before/after is named a usage pattern.

In practice, usage patterns of API methods are often not well documented. Thus, it is often challenging for developers to effectively and efficiently (re)use an API method in different contexts to complete different tasks, especially for those developers with relatively little knowledge of an API method. In a survey conducted at Microsoft in 2009, 67.6% respondents mentioned that there are obstacles caused by inadequate or absent resources for learning APIs [12]. Another field study found that a major challenge for API users is to discover the subset of the APIs that can help complete a task [13].

Based on our qualitative study with Microsoft developers, we confirmed the challenge of (re)using an API method in different contexts to complete different tasks, and were motivated to develop a tool for mining API usage patterns to effectively assist developers in practice. We further found three important aspects for developers that would use such a tool. First, the tool should be *scalable* to work with large-scale codebases with millions of lines of code and thousands of (public and private) API methods. Based on this consideration, we decided to mine usage patterns of API methods from the invocation sequences of API methods in source code, instead of mining usage patterns from more complicated data representations such as control or data flow graphs [4][11] or partial-order graphs [1]. Second, the mined patterns should be *succinct*, i.e., the mined patterns should contain few redundant patterns so that developers do not need to review similar patterns repeatedly to find the one that they are interested in. Third, the mined patterns should achieve *high coverage* of possible usages of an API method so that developers may benefit from using the tool even when the target usage pattern is less popular in the codebase being mined.

Although approaches ([1][10][14][15][18]) have been proposed to mine usage patterns of API methods from a codebase, they lack the metrics to measure the quality of mined usage patterns from the view point of developers/users. Furthermore, our empirical study (Section II) has found that the mined API usage patterns from a representative approach MAPO [18] tend to be many and redundant, posing significant barriers for being adopted in practice.

In this paper, to address these issues, we first propose two quality metrics that measure the quality of mined usage patterns of API methods from the view point of developers/users, including the succinctness and coverage of usage patterns. We further propose a novel approach called Usage Pattern Miner of API methods (UP-Miner) that mines API usage patterns from source code. UP-Miner aims to achieve high coverage and succinctness of the mined patterns. It implements the BIDE algorithm [15] to mine frequent closed API-method invocation sequences and includes a two-step clustering strategy before and after BIDE to identify usage patterns. Given a user-specified API method, UP-Miner can automatically search for all usage patterns of an API method and return associated code snippets as reuse candidates.

This paper makes the following contributions:

- We propose two quality metrics that measure the quality of the mined usage patterns of API methods, including

*: The work has been done during these authors' visit at Microsoft Research Asia.

the succinctness and coverage. Our work is the first attempt to measure the quality of mined usage patterns of API methods from the view point of developers/users.

- We propose an approach called UP-Miner for mining succinct and high-coverage usage patterns of API methods from source code. UP-Miner includes a two-step clustering strategy before and after mining frequent closed API-method sequences from source code to effectively reduce redundancy and improve the succinctness of the mined API usage patterns.
- We evaluated UP-Miner on a large-scale Microsoft codebase. The evaluation results show that UP-Miner is effective in addressing developers' API queries. UP-Miner also reduces the average percentage of redundant patterns to 11.92% (compared to 51.12% by MAPO [18]).
- We conducted a user study with Microsoft developers. The results show that UP-Miner can effectively help developers reuse APIs in practice.

The rest of the paper is organized as follows. Section II briefly introduces our motivating studies. Section III provides the problem definition. Section IV introduces the proposed UP-Miner approach. Section V and VI describe our in-house evaluations and user studies, respectively. Section VII discusses the results, and Section VIII discusses related research. Section IX concludes this paper.

II. Motivating Studies

To understand the challenges posed by API usage pattern mining and to understand whether existing methods can address the practical requirements, we conducted two empirical studies. The first one attempts to understand the characteristics of API usage patterns in a large-scale commercial codebase at Microsoft. The second one is a study on the quality of API usage patterns mined by a representative approach MAPO [18] from the same codebase.

In particular, we conducted the two studies against a large-scale codebase of a Microsoft product. We refer to it as codebase M in the rest of this paper for simplicity. Codebase M consists of over 8.5 million lines of code written in C# and over 40K API methods in total. More than one thousand developers have been involved in the development of this product over the past five years.

A. An Empirical Study of API Usage Patterns

We analyzed the characteristics of usage patterns of ten popular .NET API methods for database operations over codebase M. These ten API methods are listed in the right part of Table II. We choose these ten API methods since database operations are an important type of widely conducted operations in modern online service systems and enterprise applications. In addition, each of these API methods has a number of usage patterns for completing different database-operation tasks, such as reading/writing data from a database.

We first identified all possible usage patterns of each API method by consulting the Microsoft Development Network (MSDN) and reference books. The usage patterns were first identified by one author A and then verified by two other authors independently. The two agreed with over 90% usage patterns identified by A, and they got agreements against with the remaining patterns after minor adjustments. From codebase M, we then collected the code snippets that contain those API usage patterns. The manually identified API usage patterns and the associated code snippets in codebase M served as the "golden set" for evaluations.

We found that the distribution of API usage patterns is quite imbalanced - some API usage patterns occur very frequently and others occur quite rarely. Table I summarizes the results of the empirical study. For the ten studied API methods, the number of usage patterns ranges from 1 to 21, with an average of 6.3. The occurrence of a usage pattern ranges from 1 to 880, with an average of 71.4.

Table I. The distribution of API usage patterns in codebase M

	#Total	Min	Max	Mean	Std.Dev.
Usage Patterns	63	1	21	6.3	6.3
Pattern Occurrences	4499	1	880	71.4	252.2

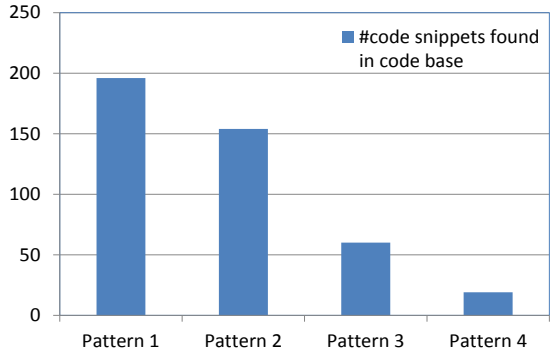

Fig.1. The usage distribution of the API method SqlCommand.ExecuteReader in codebase M

As an example, Figure 1 shows that for API method `SqlCommand.ExecuteReader`, four different usage patterns were identified. Different usage patterns have different occurrences in codebase M. Pattern 1 appears 196 times, while Pattern 4 appears only 19 times.

This result reveals that a good miner for API usage patterns should be able to discover all possible API usage patterns, even though some are less frequently used (i.e., their occurrence frequency is small). Without less frequently used patterns available in the mining result, developers would lose the opportunity to leverage the existing code to implement similar functionalities that are enabled by such usage patterns. A less frequently used pattern in the past would not necessarily also be rarely used in the future.

B. A Replication Study of a Representative Approach

As discussed in the introduction section, based on the consideration of tool scalability, we decided to mine API usage patterns from the invocation sequences of API methods in a codebase, instead of more complicated data

representations, such as data/control flow graphs or partial order groups of API method invocations. We found that MAPO [18] is a representative approach that mines API usage patterns from invocation sequences of API methods. Therefore, we conducted a study to examine the quality of API usage patterns mined by MAPO against codebase M.

We implemented MAPO according to its description [18]. Then we used MAPO to query the usage patterns of the same ten API methods as those used in the first empirical study. We found that MAPO produced a large number of patterns[1]. The number of patterns returned for each API method ranged from 14 to 94, with an average of 45.2. We manually examined each returned pattern and found that there was a large percentage of redundancy. As an example, Figure 2 shows part of the results returned by MAPO for API method `SqlConnection.Open`. We can see that many patterns are very similar: some sequences are subsequences of the others. Therefore, most of them are redundant and incur extra effort for developers in finding usage patterns of interest. The percentage of duplicated patterns for API method SqlConnection.Open was 54.26% (*Duplication* is defined in section V.B): 51 patterns (out of 94) are duplicated and should be merged with other patterns. This result implies that in order to find API usage patterns of interest from the results returned by MAPO, a user has to browse a long list of sequences (possibly with high redundancy) before she can identify the usage patterns and their associated code snippets.

```
SqlConnection.CreateCommand
SqlConnection.Open
----------------------------------------
SqlConnection.CreateCommand
SqlConnection.Open
SqlCommand.ExecuteReader
----------------------------------------
SqlConnection.CreateCommand
SqlConnection.Open
SqlCommand.ExecuteReader
SqlDataReader.Read
----------------------------------------
SqlConnection.Open
```

Fig.2. Partial list of patterns returned by MAPO when querying SqlConnection.Open

III. Problem Definition

Our empirical studies described in the previous section reveal a technical challenge: how to mine high quality API usage patterns? We measure the quality of mined API usage patterns from two main aspects from the view point of the developers/users:

- Coverage: Patterns returned by an API usage miner should be able to cover all possible usage scenarios of an API method, including less commonly-used ones.
- Succinctness: Patterns returned by an API usage miner should be succinct, since the developers usually prefer to examine only a small number of patterns. Ideally, each typical usage scenario is represented by only one pattern and there is no redundancy among the patterns.

Our definition of the problem of mining API usage patterns is as follows:

Definition (API Usage Mining): *Given a set of API method sequences and a minimum support threshold min_sup, the problem of API usage mining is to find an optimal number of k patterns, while the occurrence of each pattern is no less than min-sup, and both coverage and succinctness of the patterns can be maximized.*

IV. The Proposed Approach

To address the problem defined in Section III, we propose UP-Miner, an approach and support tool for mining succinct and high-coverage API usage patterns from source code.

A. Mining Frequent API Usage Patterns

Figure 3 shows the overall workflow of UP-Miner. For a set of call sequences that include an API method, UP-Miner first performs clustering based on the similarity of the sequences. It then mines API usage patterns from each cluster using a frequent closed sequence mining algorithm, e.g., BIDE [15], and performs clustering again to group the frequent closed sequences into patterns.

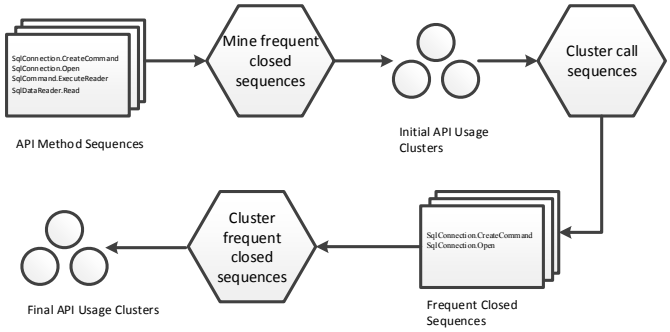

Fig. 3. Overall workflow of UP-Miner

1) Clustering API Method Sequences: We first cluster all the input sequences before identifying the frequent ones. This helps to avoid the problem of imbalanced usage distribution that we described in Section II.A: for some very common API usages, their sequences are frequent and may gain large support, while for some less common API usages, their frequency is low and thus may not reach the minimum support threshold. Clustering the API method sequences before mining frequent patterns increases the probability of the less frequent patterns to be mined out, because the less frequent patterns need to satisfy only the minimum support within a cluster instead of the entire set of method sequences.

To cluster the sequences, we propose *SeqSim*, an n-gram based technique to compute the similarity of any two sequences. We first define the n-gram set $G(X)$ for a sequence $X(x_1 x_2 \dots x_n)$ as a collection of unigrams, bi-grams, … , n-gram of X:

$$G(X) = \{x_1, x_2 \dots, x_n,\ x_1x_2, x_2x_3 \dots, x_{n-1}x_n, \dots,$$
$$x_1x_2..x_{n-1}, x_2x_3..x_{n-1}x_n, x_1x_2..x_{n-1}x_n\}$$

[1] Detailed results can be found on the project web site http://research.microsoft.com/UP-Miner

We define the similarity of two API method sequences based on the following two intuitions: (1) Two sequences are similar if they share a relatively high percentage of items (a.k.a., API method) between their corresponding n-gram sets; (2) Two method sequences are more similar if they share longer common and consecutive subsequences instead of shorter ones. We compute the similarity of two call sequences s_1 and s_2 as follows:

$$SeqSim(s_1, s_2) = \frac{\sum_i Weight(g_\cap^i)}{\sum_i Weight(g_U^i)}$$

where $g_\cap^i \in G_\cap = G(s_1) \cap G(s_2)$, $g_\cup^i \in G_\cup = G(s_1) \cup G(s_2)$,$Weight(g_\cap^i)$ is equal to the length of $g_\cap^i$.

As an example, given s_1 = *abc* and s_2 = *cab*, then we will have $G_\cap = \{a, b, c, ab\}$, $G_\cup = \{a, b, c, ab, bc, ca, abc, cab\}$, and $Weight(ab) = 1$,$Weight(ab) = 2$, $SeqSim(s_1, s_2) =$ 0.33.

Based on the SeqSim values, we cluster API method sequences. We adopt a conservative technique (e.g., the complete linkage technique [6]) for clustering: the distance between two clusters is computed as the maximum distance between a pair of items in two clusters. The distance between any two clusters should be bigger than a threshold *α*. We describe the derivation of an optimal *α* value in Section IV.B.

2) Mining Frequent Closed Sequences: To identify the frequent API usages, we implemented the BIDE algorithm [4] to mine frequent closed call sequences. There are a number of key concepts: a sequence s_1 is called a sub-sequence of sequence s_2 and s_2 is called a super-sequence of s_1 if sequence s_1 is contained in sequence s_2; A sequence s is a frequent sequence in a sequence set if $sup(s) \geq min\_sup$, where $sup(s)$ is the support of sequence s, i.e., the ratio of the sequences in the sequence set that are super-sequences of s, min_sup is the minimum support threshold; s is a frequent closed sequence (as defined in [6]) if sequence s is frequent and there are no proper super-sequence of s with the same support.

We apply BIDE to each cluster derived from the process described in the previous section and produce frequent closed sequences for the cluster. For example, for the three sequences in a cluster, *ab*, *abc*, and *abd*, if the min_sup is 0.5, BIDE produces frequent closed sequences *ab*. While other frequent sequence miners such as Bitmap (the sequence mining algorithm used in MAPO) return *a*, *b*, and *ab*. Clearly, since *ab* is frequent, *a* and *b* are also frequent and need not to be listed. BIDE returns only the longest sequences if there are subsequences with the same support.

3) Clustering Frequent Closed Sequences: After the first clustering described in Section IV.A.1, sequences that are not similar are divided into different clusters. However, the frequent closed sequences mined from different clusters could still be similar, causing redundancy among the resulting patterns. To consolidate the frequent closed patterns mined from different clusters, we perform another round of clustering and treat each resulting cluster as a usage pattern. For example, consider a situation in which the frequent closed sequence abc and ab are mined from different clusters. After clustering, these two sequences are grouped together as one pattern, and is shown as abc. Using this technique, we further reduce the number of redundant sequences in the mined API usage patterns. In this round of clustering, we again use the complete-linkage clustering technique with a threshold β (we describe the derivation of an optimal β value in Section IV.B). The similarity between two frequent sequences is also computed using the SeqSim metric described in Section IV.A.1.

B. Determination of the Optimal Number of Patterns

We adopted the two-step clustering strategy described in the previous section to improve the quality of mined patterns. Such clustering can help identify less common API usages whose frequency does not satisfy the minimum support of the entire sequences. The two-step clustering process involves two coefficients α (the threshold for the pre-BIDE clustering) and β (the threshold for the post-BIDE clustering), whose values vary independently. Different combinations of α and β can result in different clustering performances and different numbers of patterns and thus affecting the quality of mined usage patterns.

We propose an automatic technique to determine the optimal threshold values. As described in Section III, the goal of API usage mining is to identify a number of patterns from a set of method sequences in order to maximize both the coverage and the succinctness of the patterns.

Measuring the number of resulting patterns can reflect succinctness. In order to ensure that one usage scenario is covered by a minimum number of mined usage patterns, we expect that the mined patterns are dissimilar as much as possible from one another. To measure to what extent the mined usage pattern are dissimilar to each other, we define an overall dissimilarity metric *D* (the dissimilarity of the resulting pattern graphs). The metric *D* is defined as the average dissimilarity between all pairs of usage patterns:

$$D = \frac{\sum_{i=0}^{n-1} \sum_{j=i+1}^{n} Dsim(i,j)}{n*(n-1)/2}$$

where $Dsim(i, j)$ is the dissimilarity of usage pattern *i* and *j*. $Dsim(i, j)$ is defined as $(1 - size(BG_i \cap BG_j)/size(BG_i \cup BG_j))$, BG_i is the set of distinct bi-grams of all frequent closed sequences in usage pattern *i*. This definition is based on the intuition that the usage patterns will be more dissimilar if the percentage of common bi-grams that appear in the frequent closed sequences of these two usage patterns is lower.

The selection of the two clustering thresholds should produce patterns that maximize the dissimilarities among identified patterns and minimize the number of resulting patterns. Therefore, we define the following utility function *U*:

$$U = D/n$$

We propose a search-based algorithm that exhaustively searches for the optimal combination of threshold values that maximize the utility function (Figure 4). In Figure 4, we first set the pre-BIDE clustering threshold α to a maximum value α_{max} and the post-BIDE clustering threshold β to β_{max}. We compute the utility function for each pair of clustering thresholds, and adjust the threshold values by decreasing it value by a step δ, to search for the maximum utility value, until their values reach α_{min} or β_{min}. Finally, we return the optimal threshold values, m_α and m_β, that lead to the maximum utility value. Based on our empirical exploration, we set $\alpha_{max} = \beta_{max} = 1.00$, $\alpha_{min} = \beta_{min} = 0.00$, and $\delta = 0.01$.

C. Tool Implementation

UP-Miner includes an API parser to parse the source code files and to create an index for each API method. The API parser is based on Roslyn[2], which constructs AST trees for each source code file. We also filter out some common logging and assertion methods such as `Log.Info` and `Assert.AreEqual`, since these methods are typically irrelevant to the program logic. After identifying API methods from their call sites in the source code files, we then create an index file for each API method and collect API method sequences. UP-Miner then mines the API usage patterns and presents the resulting patterns as probabilistic graphs, which are ranked by the number of occurrences. Besides the graphical representation of a usage pattern, UP-Miner also provides developers the code snippets associated with the pattern. We briefly present how we build the probabilistic graph for a usage pattern and our prototype in the following:

1) Deriving Probabilistic Graphs: In our tool implementation, we represent the API usage patterns as a probabilistic graph to help users intuitively examine and understand the patterns. A graph graphically represents a cluster of frequent closed patterns described in the previous section. Developers can understand the usage scenarios associated with an API method by examining the graph. A node in the graph indicates an API method and an edge

```
DetermineOptimalParameters
1.  set the initial pre-BIDE clustering threshold α to
2.  set max = 0,  = 0, = 0
3.  while α >
4.  set the initial post-BIDE clustering threshold β  to
5.      while β >
6.          compute the utility function U
7.          If U is greater than max
8.              max = U,  = α , = β
9.          EndIf
10.     decrease β  by δ
11.     EndWhile
12. decrease α by δ
13. EndWhile
14. return ,
```

Fig.4. The algorithm for determining the optimal threshold values

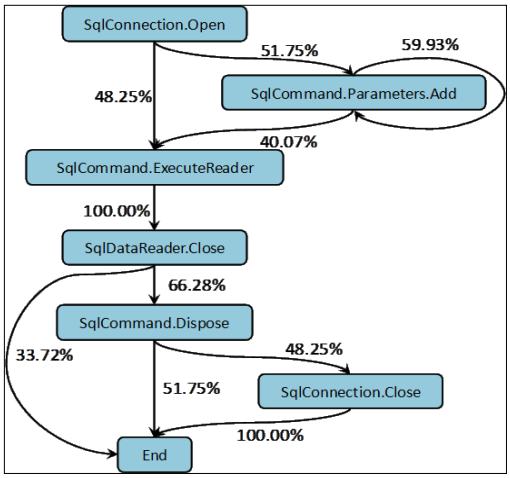

Fig. 5. An example usage pattern of SqlConnection.Open

indicates a temporal relationship between two methods. Each edge is associated with a probability value that indicates the probability of calling one method after the other among all occurrences of this usage pattern in the codebase. Figure 5 shows an example of a graphical representation of a usage pattern for `SqlConnection.Open`. By examining the graph, users can better understand usage scenarios associated with this API method.

Note that there are also other graphical presentations that can be used to present usage patterns such as finite state automata. In the future, we will further study which graphical representation best fits our needs.

2) Prototype: Figure 6 shows a screenshot of UP-Miner. On the top of the window is an input box, which allows developers to input an API method's name. On the left side of the window is a list box that lists all patterns mined by UP-Miner. The graph panel shows the pattern as a probabilistic graph and the code panel shows the associated code snippets.

V. Experiments

A. Experimental Design

In order to evaluate our approach, we conducted a number of experiments against the Microsoft codebase M, which is a large codebase described in Section II. We selected 20 .NET API methods in our experiments (as shown in Table II). Ten of them are the same as the ones used for our empirical studies in Section II, and the other ten are http-request handling API methods. We choose these API methods because they are two important types of widely used .NET framework API methods in modern online service systems and enterprise applications. Through our experiments, we intend to investigate the following three research questions:

RQ1: How effective is UP-Miner in generating succinct and high-covering API usage patterns?

[2] http://msdn.microsoft.com/en-gb/roslyn

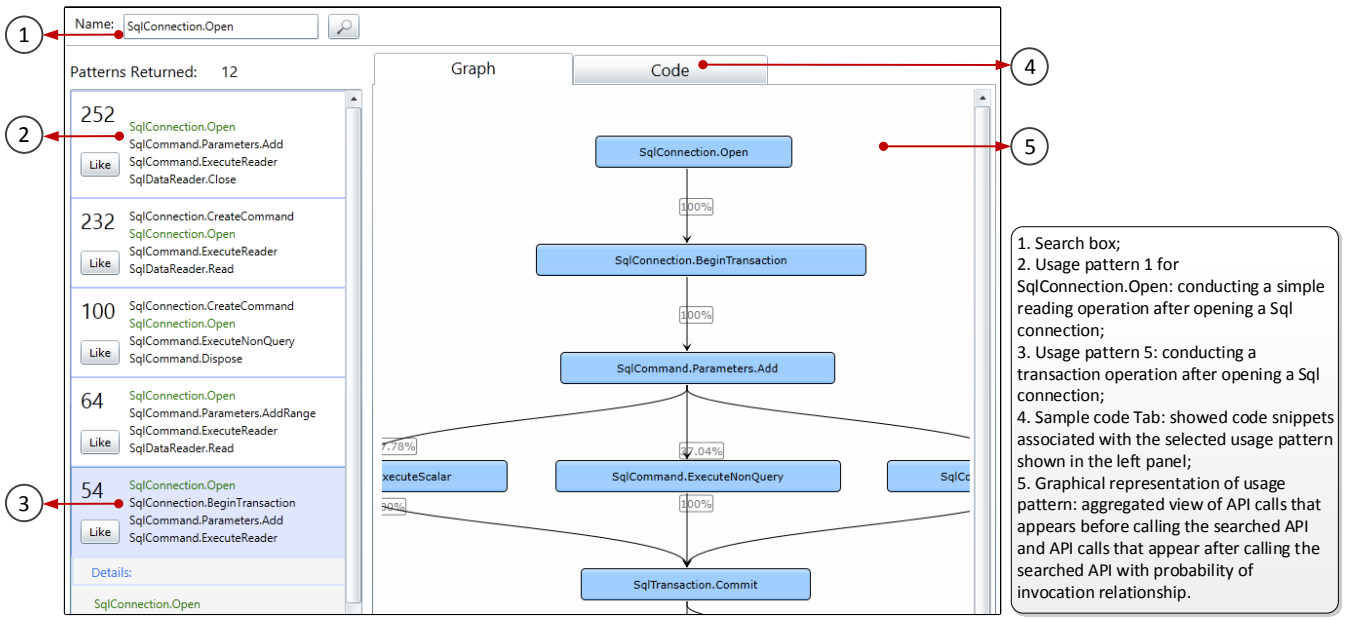

Fig 6. Screenshot of UP-MINER prototype

This research question evaluates the effectiveness of UP-Miner in generating succinct and high-coverage API usage patterns. To answer this question, we manually identified a "golden set" of API usage patterns for each of these 20 API methods, as described in Section II.C. After collecting the golden set, we then ran queries for each API method against both UP-Miner and MAPO, and compared the results. In Section IV.B, we describe the metrics (used for the result comparison) to reflect the succinctness and coverage of the mined usage patterns.

RQ2: How much benefit can the two-step clustering process bring?

In Section IV, we proposed a two-step clustering process in the design of UP-Miner: one before the frequent closed sequence mining and one after. This research question evaluates how much benefit such a two-step clustering process brings by comparing the performance of UP-Miner (with such a process) and a BIDE-only approach (without such a process).

RQ3: How much better is UP-Miner compared with the one-clustering+BIDE approach?

Related to RQ2, this RQ further evaluates how much benefit the second clustering brings, by comparing the API usage patterns mined by UP-Miner and by the one-clustering+BIDE approach (without the second clustering).

B. Metrics

For RQ1 and RQ3, we define a variant of the Purity and Inverse Purity metrics [9] for evaluating the effectiveness of the tools in producing succinct and high-covering patterns.

We denote C as the set of clusters to be evaluated. A cluster represents a mined usage pattern. We denote L as the set of patterns in the golden set. $|L \text{ in } C_i|$ represents the number of the patterns defined in L and can be found in the ith cluster C_i. $|L_i \; in \; C|$ represents the number of patterns in C that contains the ith pattern L_i in the golden set. We define the Average Purity (AP) and Average Inverse Purity (AIP) metrics as follows:

$$AP = \frac{1}{|C|}\sum_{i=1}^{|C|}(|\text{L in } C_i| == 0 ? 0 : \frac{1}{|\text{L in } C_i|})$$

$$AIP = \frac{1}{|L|}\sum_{i=1}^{|L|}(|L_i \; in \; C| == 0 ? 0 : \frac{1}{|L_i \; in \; C|})$$

The values of the metrics range from 0 to 1, the higher the better. Ideally, each pattern produced by an API usage miner should represent only one pattern in the golden set (thus AP is 1), and each pattern in the golden set should be represented by only one pattern produced by an API usage miner (thus AIP is 1). Higher AIP values indicate higher coverage and (or) lower redundancies, since the AIP value of API pattern miner A would be lower than that of B if a golden set pattern does not appear in any patterns returned by A, but appears in one returned by B; the AIP value of A could also be lower than that of B if gold set patterns are represented by a fewer number of mined patterns returned by A compared that returned by B. Higher AP values mean that the mined usage patterns are more succinct in terms of a fewer number of golden set patterns appearing in one mined usage pattern. We define *F-measure* to compute the weighted average of Average Purity and Inverse Purity to get a robust and balanced quality measurement. The value of *F-measure* is from 0 to 1, the higher the better.

$$F - Measure = \frac{2 \times AP \times AIP}{AP + AIP}$$

To further directly measure the effectiveness of the tools in producing succinct patterns, we design the metric *Duplication* as follows:

$$Duplication = \#Duplicated/\#Total,$$

where *#Total* represents the number of patterns returned by the tools and *#Duplicated* represents the number of duplicated patterns. To calculate *#Duplicated*, we first find a minimal set of mined usage patterns that can represent as many golden-set patterns as possible, or maybe even all of them; the number of remaining mined patterns is defined as #Duplicated. The values of the metric range from 0 to 1, the lower the better.

For RQ2, because BIDE does not produce clusters, the Average Purity and Average Inverse Purity metrics are not applicable. We used the metrics *Recall*, *Precision*, and *F-measure* to measure the effectiveness of BIDE. $|L \text{ in } C|$ represents the number of the patterns defined in L that can be found in C. $|C \text{ in } L|$ represents the number of patterns in C that can be found in L.

$$Recall(L,C) = \frac{|L \text{ in } C|}{|L|} \quad Precision(L,C) = \frac{|C \text{ in } L|}{|C|}$$

$$F-measure(L,C) = \frac{2 \times Recall(L,C) \times Precision(L,C)}{Recall(L,C) + Precision(L,C)}$$

The values of the metrics are from 0 to 1, the higher the better.

C. Experimental Results

We present our experimental results that answer the three research questions and compare the performance of UP-Miner with a number alternative approaches in this section.

1) RQ1: How Effective Is UP-Miner in Generating Succinct and High-Coverage API Usage Patterns? Table II shows the evaluation results of UP-Miner and MAPO on 20 APIs using the metrics F-measure and Duplication. In terms of F-measure, UP-Miner outperformed MAPO on 17 APIs (85%). The improvement ranged from 0.359 to 0.714, with an average of 0.285. In terms of Duplication, UP-Miner performed much better than MAPO on all 20 APIs, with an average of 11.92% (compared to MAPO's 51.12%). Paired t-tests confirmed that the results were statistically significant at the significance level 0.05. The evaluation results show that UP-Miner is effective in mining succinct and high-covering API usage patterns.

2) RQ2: How Much Benefit Can the Two-Step Clustering Process Bring? Table III shows the comparison results (F-measure), which were computed through Recall and Precision between BIDE and UP-Miner (pre-BIDE clustering + BIDE + post-BIDE clustering). For 19 APIs, UP-Miner achieved a better or equal performance than/to BIDE, and the improvement ranged from 0 to 0.603 with an average of 0.173. A paired t-test shows that the results are statistically significant at the significance level 0.05. The results confirm that the proposed two-step clustering is highly beneficial in terms of improving the quality of the mined API usage patterns.

3) RQ3: How Much Better Is UP-Miner Compared with the One-Clustering + BIDE Approach? Table IV shows the comparison results (F-measure of AP and AIP) between UP-Miner and one-clustering + BIDE. The results show that UP-Miner performed better than clustering + BIDE on 16 out of 20 API methods, with an average improvement of 0.426. A paired t-test shows the results are statistically significant at the significance level 0.05. The results confirm that the proposed two-step clustering is more effective than single one-step clustering.

4) Comparing with Alternative Approaches: Table V summarizes the results of our experiments. Among 20 studied API methods, UP-Miner performs better than One-clustering + BIDE, MAPO, and BIDE approaches on 15, 17, and 16 API methods, respectively. The average improvement is 0.173, 0.285, and 0.426, respectively. The experimental results show that, in general, UP-Miner outperforms MAPO, a representative approach, and two alternative approaches.

Table II. Evaluation Results (Comparisons of API usage pattern mining result between UP-Miner and MAPO)

API Name	Tool	# patterns returned	F-measure	Duplication	API Name	Tool	# patterns returned	F-measure	Duplication
WebRequest. Create	MAPO	71	34.76%	57.75%	SqlConnection. Open	MAPO	94	39.46%	54.26%
	UP-Miner	8	47.66%	12.00%		UP-Miner	12	33.10%	0.00%
HttpWebRequest. GetResponse	MAPO	58	16.07%	70.69%	SqlCommand. ExecuteNonQuery	MAPO	53	18.56%	60.38%
	UP-Miner	7	52.42%	14.00%		UP-Miner	5	61.54%	0.00%
HttpWebResponse. GetResponseStream	MAPO	46	40.92%	63.04%	SqlCommand. Parameters.Add	MAPO	56	19.28%	53.57%
	UP-Miner	9	50.00%	11.11%		UP-Miner	8	25.00%	12.50%
HttpWebRequest. Create	MAPO	22	32.52%	40.91%	SqlCommand. ExecuteReader	MAPO	57	28.96%	68.42%
	UP-Miner	4	54.91%	0.00%		UP-Miner	4	50.00%	25.00%
HttpWebRequest. Headers.Add	MAPO	11	21.27%	45.45%	SqlDataReader. Read	MAPO	67	3.95%	61.19%
	UP-Miner	3	60.00%	33.00%		UP-Miner	13	38.01%	30.77%
WebRequest. GetResponse	MAPO	6	24.55%	58.82%	SqlConnection. Close	MAPO	44	30.03%	34.09%
	UP-Miner	6	54.13%	0.00%		UP-Miner	2	55.24%	0.00%
HttpWebRequest. GetRequestStream	MAPO	6	49.07%	33.33%	SqlCommand. ExecuteScalar	MAPO	29	34.44%	65.52%
	UP-Miner	1	30.00%	16.67%		UP-Miner	2	50.00%	0.00%
HttpResponse. SetStatusText	MAPO	6	28.58%	83.33%	SqlConnection. CreateCommand	MAPO	19	27.17%	42.11%
	UP-Miner	1	100.00%	0.00%		UP-Miner	1	30.77%	0.00%
HttpResponse. SetContent	MAPO	6	27.91%	83.33%	SqlCommand. Parameters. AddWithValue	MAPO	14	25.00%	35.71%
	UP-Miner	2	66.67%	50.00%		UP-Miner	3	50.00%	33.33%
HttpResponse. Write	MAPO	3	N.A.	0.00%	SqlDataAdapter. Fill	MAPO	19	9.09%	10.53%
	UP-Miner	1	66.67%	0.00%		UP-Miner	1	N.A.	0.00%

Table III. Comparisons between BIDE and UP-Miner (F-Measure)

APIName	BIDE	UP-MINER	APIName	BIDE	UP-MINER
WebRequest.Create	19.51%	69.23%	SqlConnection.Open	13.33%	57.14%
HttpWebRequest.GetResponse	38.46%	90.91%	SqlCommand.ExecuteNonQuery	N/A	68.57%
HttpWebResponse.GetResponseStream	20.00%	57.14%	SqlCommand.Parameters.Add	N/A	33.33%
HttpWebRequest.Create	28.57%	88.89%	SqlCommand.ExecuteReader	N/A	60.00%
HttpWebRequest.Headers.Add	66.67%	80.00%	SqlDataReader.Read	80.00%	70.00%
WebRequest.GetResponse	16.67%	50.00%	SqlConnection.Close	11.11%	71.29%
HttpWebRequest.GetRequestStream	24.39%	40.00%	SqlCommand.ExecuteScalar	N/A	50.00%
HttpResponse.SetStatusText	100.00%	100.00%	SqlConnection.CreateCommand	28.29%	44.44%
HttpResponse.SetContent	100.00%	100.00%	SqlCommand.Parameters.AddWithValue	N/A	50.00%
HttpResponse.Write	28.57%	66.67%	SqlDataAdapter.Fill	N/A	50.00%

Table IV. Comparisons between One-Clustering + BIDE and UP-Miner (F-Measure)

APIName	Clustering + BIDE	UP-MINER	APIName	Clustering + BIDE	UP-MINER
WebRequest.Create	52.30%	47.66%	SqlConnection.Open	53.83%	33.10 %
HttpWebRequest.GetResponse	38.46%	90.91%	SqlCommand.ExecuteNonQuery	52.57%	61.54 %
HttpWebResponse.GetResponseStream	41.84%	50.00%	SqlCommand.Parameters.Add	16.00%	25.00 %
HttpWebRequest.Create	52.01%	54.90%	SqlCommand.ExecuteReader	44.22%	50.00 %
HttpWebRequest.Headers.Add	32.14%	60.00%	SqlDataReader.Read	14.86%	22.58 %
WebRequest.GetResponse	44.44%	50.00%	SqlConnection.Close	43.59%	55.24 %
HttpWebRequest.GetRequestStream	24.39%	40.00%	SqlCommand.ExecuteScalar	25.64%	50.00 %
HttpResponse.SetStatusText	40.00%	100.00%	SqlConnection.CreateCommand	52.00%	30.77 %
HttpResponse.SetContent	50.00%	66.67%	SqlCommand.Parameters.AddWithValue	36.36%	50.00 %
HttpResponse.Write	33.33%	66.67%	SqlDataAdapter.Fill	20.00%	n/a

VI. USER STUDY

We designed two user studies (one controlled study and one user survey) to explore the usefulness of our tool and the user experiences.

Table V. Summary of comparisons

	One-clustering + BIDE	MAPO	BIDE
# API UP-MINER Wins	15	17	16
Average F-measure Improvement	0.173	0.285	0.426

A. Controlled Study

In the controlled study, we created a program under development for collecting, displaying, and saving information from/to a text file. We then purposely left three major functionalities to be completed by the participants. These three tasks cover three different programming aspects: Task 1 concerns database operations, Task 2 designs the user interface, and Task 3 deals with XML file processing. The details of these tasks are as follows:

Task 1: Reading information. This task asked the participants to complete a method that reads information from a text file, stores the information in a database, and retrieves information given search conditions. The database should roll back to the original status if the operations fail. We provided the API method `SqlConnection.Open` as the starting point to the participants and asked them to complete the rest of the code. In this task, two types of API usages were involved: the transaction-related database operations (implemented via *`SqlConnection.Open-SqlConnection.BeginTransaction-SqlCommand.ExecuteNonQuery`*) and the normal database operations (implemented via *`SqlConnection.Open-SqlCommand.ExecuteReader`*). The former is less frequently used than the latter. Therefore, in this task, the participants had opportunities to select either more popular or less popular API usages.

Task 2: Displaying information. In this task, the participants were asked to depict information as a bar chart. We provided the API method *`Graphics.DrawRectangle`* as the starting point for this task.

Task 3: Writing information. This task required the participants to save the information in an XML file. We provided the API method *`XmlWriter.Create`* as the starting point for this task.

We designed a cross evaluation: we invited six participants (interns working in our lab on other teams) and divided them into three groups; we then asked each group to implement the three programming tasks. Each group had the same average number of years of C# experience. The C# programming experiences of the participants ranged from 1 to 18 months. The group arrangement is shown in Table VI. We asked the participants to implement the three tasks using Koders[3], MAPO, and UP-Miner, respectively. After the participants finished each of the tasks, we examined the correctness of the task completion.

Table VI. Group arrangement of the controlled user study

	Koders	MAPO	UP-Miner
Task 1	P1 P2	P3 P4	P5 P6
Task 2	P3 P4	P5 P6	P1 P2
Task 3	P5 P6	P1 P2	P3 P4

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

Table VII shows the results returned by the participants. Using Koders and MAPO, the participants searched 16-17 API queries, but did not manage to finish all three tasks on time (we asked participants to complete all tasks in 2 hours). Using UP-Miner, they completed all the tasks with 13 API queries on average. The study showed that the UP-Miner tool can help developers finish more tasks with fewer API queries. After the study, the participants explained that the reason for the incomplete tasks was that they did not obtain useful API usage patterns by using Koders or MAPO in many cases, especially for those usage patterns that are less popular, e.g., *SqlConnection.BeginTransaction*.

Table VII. Results of the controlled user study

	Koders	MAPO	UP-Miner
#Tasks Completed	1	2	3
#Searched API	16	17	13

B. User Survey

We also conducted a survey of Microsoft developers to investigate whether UP-Miner helps with their programming tasks. 60 developers working on codebase M participated in the survey. In addition to asking participants to experience the UP-Miner tool, we also asked them whether they agree on the usefulness of the tool. The scores were given on a five-point Likert scale (Strongly agree - 5, Agree - 4, Neutral - 3, Disagree - 2, and Strongly Disagree - 1).

We received feedback from 28 out of the 60 developers who we asked. All of them gave positive comments on the UP-Miner tool and believed that such a tool could help with their programming work. The average Likert score was 4.28 (with standard deviation of 0.60). The developers found that the returned patterns, the graphical representation, and the sample code can help them reuse APIs. Below are some comments and feedback from the developers:

Developer 1: *Cool! This tool is really what I am looking for to find the right sample code.*

Developer 2: *This is really a great tool. Especially I love the Pattern and the graph features, especially the graph feature. This graph could show the usage pattern and context, it provides much more meaningful interpretation than plain text, this will help understanding other's code a lot.*

Developer 3: *It is really a cool idea and tool to help understand the API usage, especially for some internal library which doesn't have too many docs to tell you how to use the API.*

VII. Discussion

A. Why Does UP-Miner Work?

We have identified the following reasons that UP-Miner outperforms existing methods, especially MAPO.

- Guiding the algorithm design by two quality metrics. We proposed the succinctness and high coverage as two quality metrics of mined usage patterns from a developer's perspective. This is a key difference between UP-Miner and existing approaches.
- Two-step clustering strategy. In our method, we perform two-step clustering before and after the mining of frequent sequences. After the first round of clustering, some popular usage patterns are put in bigger clusters, and less popular usage patterns are put in relatively small clusters. This ensures that even less popular usage patterns will be mined using a BIDE algorithm with a ratio threshold. The second round of clustering further helps us to reduce the redundant patterns that possibly appear in the frequent closed sequence set mined from different clusters (generated by the first round of clustering). Our evaluations have shown that such a strategy can improve the quality of the patterns.
- Similarity measures. In our work, we propose SeqSim, an n-gram based technique to compute the similarity of two API method sequences, considering not only the occurrences of each API method, but also the occurrences of n-gram API methods. It also takes into consideration the weight of each n-gram of API methods.
- Mining frequent closed sequences. As described in Section II, MAPO uses the Bitmap algorithm to mine frequent sequences, resulting in a large number of redundant sequences. In contrast, our approach includes the BIDE algorithm to mine frequent closed sequences, thus reducing the redundancy of the resulting sequences.

B. Threats to Validity

We identify several threats to validity.

In our study, we use codebase M as the experimental subject, which is a large-scale industrial project. However, some codebases, especially codebases of small projects, may not contain enough API instances for UP-Miner to mine.

We selected 20 .NET API methods in our experiments. Although these API methods are well-known methods covering both database and Web domains, they are still limited in number.

Our empirical study involves human subjects. The limited number and the programming capabilities of the human subjects may bias the results. To reduce this threat, we used a controlled study and a crossover design. In the future, we plan to conduct experiments and user studies involving more subjects, API methods, and programming tasks to further reduce this threat.

VIII. Related Work

A. Code Search

Mining API usage patterns is closely related to the work on code search and specification mining. Koders and Google code search are code search engines that can return code snippets containing the keywords (or Regular Expressions) of API method names. Strathcona [8] is a code snippet recommender, which locates a set of relevant code snippets by matching the structure of the code under development with the code snippets in codebase. The approach proposed by Acharya et al. [1] mines API partial orders from source code files, but it does not do clustering first to group similar orders into the same clusters and different orders into different clusters. Portfolio [10] visualizes relevant functions and their usages using a combination of models that address surfing

behavior of programmers and uses PageRank to mine the relationship of functions. However, Portfolio cannot identify the orders of any two functions. Recently, Buse and Weimer [4] presented an automatic technique for mining and synthesizing human-readable documentation of program interfaces. Its algorithm is based on a combination of path sensitive dataflow analysis, clustering, and pattern abstraction. ParseWeb [14] accepts queries of the form "Source -> Destination" from a programmer and gives the code samples containing the given Source and Destination object types. Wasylkowski et al. [17] applied static analysis to mine object usage models from code. The object usage model is represented as finite state automata, which can then be used to detect anomalies. Nguyen et al. [11] developed GraPacc, which is a graph-based and pattern-oriented tool for code completion. It takes as an input a database of usage patterns and completes the code under editing based on its context and those patterns. API usage patterns can be also mined via dynamic analysis. For example, Ammons et al. [2] proposed inferring a specification by observing program execution and summarizing frequent interaction patterns as state machines.

Unlike the related methods, we adopt a two-step clustering strategy and mine frequent closed method sequences from an organization's local codebase based on an API query. We aim to produce high quality (both succinct and high-covering) API usage patterns.

B. *Frequent Pattern Mining*

A major challenge in frequent pattern mining is the sheer size of its mining results [7]. In many cases, a low minimum support may generate an explosive number of output patterns, severely restricting the usage of a frequent pattern miner. Researchers have proposed various techniques to reduce the large number of frequent patterns, while maintaining the quality of identified patterns. For example, Calders and Goethals [5] proposed mining compressed, non-derivable frequent patterns. Wang et al. [16] proposed pCluster, an algorithm to detect clusters of patterns. They consider two objects similar if they exhibit a coherent pattern on a subset of dimensions, and demonstrated its effectiveness for microarray data analysis.

In our work, we apply BIDE and propose a two-step clustering strategy for producing high-quality API usage patterns. We also propose an optimization technique that maximizes both coverage and succinctness of the resulting patterns.

IX. Conclusion

In this paper, we have proposed UP-Miner, an approach and support tool that mines API usage patterns from source code. Our approach considers the coverage and the succinctness of the mined patterns. In our implementation, we further present a usage pattern as a graph to facilitate developers easily understanding the mined usage patterns. We perform evaluations on a large-scale Microsoft codebase and the results have shown the effectiveness of the proposed approach and our approach outperforms an existing representative approach (MAPO). The user studies performed with Microsoft developers and interns also confirm the usefulness of the proposed approach.

In the future, we plan to explore techniques of automatic code completion based on the returned API usage patterns. We also plan to transfer the UP-Miner technique to Microsoft development teams. A video demo of UP-Miner can be accessed at http://research.microsoft.com/UP-Miner.

Acknowledgement

We thank all Microsoft developers who participated in our user studies and provided us helpful comments.

References

[1] M. Acharya, T. Xie, J. Pei, J. Xu. "Mining API patterns as partial orders from source code: from usage scenarios to specification," In Proc. ESEC/FSE 2007, pp. 25-34.

[2] G. Ammons, R. Bodik, J. R. Larus. "Mining specifications," In Proc. POPL 2002, pp. 4-16.

[3] J.Ayres, J. Gehrke, T. Yiu, and J. Flannick. "Sequential pattern mining using a bitmap representation," In Proc. SIGKDD 2002.

[4] P.L. Buse and W. Weimer. "Synthesizing API Usage Examples," In Proc. ICSE, pp 782-792, 2012.

[5] T.Calders, B. Goethals, "Mining all non-derivable frequent itemsets". In: Proc PKDD'02, pp. 74-85.

[6] J. Han and M. Kamber, Data Mining: Concept and Techniques, Elsevier, 2006.

[7] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent Pattern Mining: Current Status and Future Directions. Data Min. Knowl. Discov. 15, 1, pp. 55-86, 2007.

[8] R.Holmes and G. C. Murphy. "Using structural context to recommend source code examples," In Proc. ICSE, 2005.

[9] C.D.Manning, P.Raghavan, and H.Schtze. Introduction to Information Retrieval, Cambridge University Press, 2008.

[10] C.McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usages," In Proc. ICSE 2011, pp. 111-120.

[11] A.Nguyen, T. Nguyen, H. Nguyen, A. Tamrawi, H. Nguyen, J. Al-Kofahi, and T. Nguyen. "Graph-based pattern-oriented, context-sensitive source code completion," In Proc. ICSE, 2012.

[12] M.P. Robillard, "What makes apis hard to learn? Answers from developers," IEEE Softw., vol. 26, no. 6, pp. 27–34, 2009.

[13] M.P. Robillard and R. DeLine. A Field Study of API Learning Obstacles. *Empirical Soft. Engin.*, 16(6): 703-732, 2011.

[14] S.Thummalapenta, T. Xie. "PARSEWeb: a programmer assistant for reusing open source code on the web," In Proc. ASE 2007.

[15] J.Wang and J. Han. "BIDE: efficient mining of frequent closed sequences," In Proc. ICDE 2004, pp. 79.

[16] H.Wang, W. Wang, J. Yang, and P. S. Yu. 2002. "Clustering by pattern similarity in large data sets," In Proc. SIGMOD 2002.

[17] A.Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in ESEC-FSE '07. ACM, 2007, pp. 35–44.

[18] H. Zhong, T. Xie, L. Zhang, J. Pei, H. Mei. "MAPO: mining and recommending API usage patterns," In Proc. ECOOP 2009, pp. 318-343.