Mining Multi-level API Usage Patterns

Mohamed Aymen Saied, Omar Benomar, Hani Abdeen, and Houari Sahraoui
DIRO, Université de Montréal, Montréal, Canada
{saiedmoh,benomaro,abdeenha,sahraouh}@iro.umontreal.ca

***Abstract*—Software developers need to cope with complexity of Application Programming Interfaces (APIs) of external libraries or frameworks. However, typical APIs provide several thousands of methods to their client programs, and such large APIs are difficult to learn and use. An API method is generally used within client programs along with other methods of the API of interest. Despite this, co-usage relationships between API methods are often not documented. We propose a technique for mining *Multi-Level API Usage Patterns* (MLUP) to exhibit the co-usage relationships between methods of the API of interest across interfering usage scenarios. We detect multi-level usage patterns as distinct groups of API methods, where each group is uniformly used across variable client programs, independently of usage contexts. We evaluated our technique through the usage of four APIs having up to 22 client programs per API. For all the studied APIs, our technique was able to detect usage patterns that are, almost all, highly consistent and highly cohesive across a considerable variability of client programs.**

***Keywords*-API Documentation; API Usage; Usage Pattern; Software Clustering;**

I. Introduction

Nowadays, software systems increasingly depend on external library and frameworks. Software developers need to reuse functionality provided by these libraries through their Application Programming Interfaces (APIs). Hence, software developers have to cope with complexity of existing APIs that are needed to accomplish their work.

Despite recent progress in API documentation and discovery [1], [2], [3], [4], large APIs are difficult to learn and use [5], [2]. Notably, software developers might spend considerable time and effort to identify the subset of the class's methods that are necessary and sufficient to implement (or at least to start the implementation of) their daily work. These difficulties comes from two reasons. On the one hand, the co-usage relationships between API methods are often not documented. The documentation of an API method is in most cases limited to the functionality provided by that method alone, even though an API method is usually used within client programs along with other methods of the API. On the other hand, the elaborated documentation of an API class may be very detailed, with verbos text, as it tries to specify all aspects that a client might need to know about the class of interest [3].

To learn how to use API's methods, software developers usually search for code snippets by means of source code search engines, such as the Ohloh Code search engine [6]. Yet, existing source code search engines return a huge number of code snippets that use the API's methods of interest. Therefore, we believe that identifying *usage patterns* for the API can be helpful for a better learning of common ways to use the API.

Recently, automated techniques to detect API usage properties have gained a considerable attention [7], [8], [2], [1]. These existing techniques are valuable to facilitate API understanding and usage. However, since the interference between different API usage scenarios could negatively impact the usage pattern mining process, existing techniques mainly focus on identifying API patterns in a specific usage scenario. Such techniques are proposed for recommending usage examples of the API that are relevant to one task at hand (e.g., [2]), and/or for auto completion of source code in a specific context (e.g., [9]). In consequence, they fail short in identifying API usage patterns that are, at the same time, *context independent* and *reflect different interfering API usage scenarios*.

In this paper, we propose a technique for mining *multi-level API usage patterns*. We define a multi-level API usage pattern as a distribution on different usage cohesion levels of some APIs methods which are most frequently used together in client programs, in a consistent way, and regardless of the variability of client programs. Our technique for detecting such usage patterns is based on the analysis of the frequency and consistency of co-usage relations between the APIs methods within a variety of client programs of the API of interest. The rational behind this multi-level distribution of methods in a usage pattern is to identify the pattern's core, which represents the pattern's methods that are 'always' used together, and to reflect interfering usage scenarios of the pattern's core and the rest of the APIs methods. Hence, multi-level usage patterns add a new dimension which can be used to enhance the API documentation with co-usage relationships between methods of the API of interest.

We evaluated our technique using four different APIs: HttpClient [10], Java Security [11], Swing [12] and AWT [13]. In our evaluation, we used 12 client programs of the HttpClient and Java Security APIs, and 22 client programs of the Swing and AWT APIs. We opted for a comparative evaluation of our technique MLUP against the technique of MAPO, as described in [2], [14] for identifying closed sequential usage patterns. The results show that, for every studied API, usage patterns identified by MLUP are significantly more cohesive than those identified by MAPO. Moreover, MLUP was able to identify a larger number of usage patterns than MAPO, and usage patterns identified by MLUP are overall larger and more complex than those identified by MAPO. And, in many cases, MLUP has identified API usage patterns that are present within client programs offering completely different features. Furthermore, to evaluate the generalization of identified usage patterns to potential new client programs of the API of interest, we performed leave-one-out cross validations. The results show that across a considerable variability of client programs, usage patterns identified by MLUP remain more cohesive than those identified by MAPO.

978-1-4799-8469-5/15 © 2015 IEEE

SANER 2015, Montréal, Canada

The remainder of the paper is organized as follows. Section II motivates the usefulness of this work with two actual examples from HttpClient and Swing APIs. We explain our approach in Section III and present the case study used for evaluating it in Section IV-A. Section V presents and analyzes the results of our study. In Section VI we discuss further our approach and the evaluation results. Finally, Section VII presents various contributions that are related to our work, before concluding in Section VIII.

II. Motivation Examples

In this section we present two motivation examples to illustrate the following aspects of the addressed problem: (1) the same usage pattern can be used in different contexts (we have to be context independent); and (2) interfering API usage scenarios could be presented using multi-level usage patterns.

A. HttpClient Authentication

Consider the HttpClient API [10], which provides a feature-rich package implementing the client side of HTTP standards and recommendations. For this API, each time the method `setProxy(String,int)` of `HostConfiguration` class is invoked, our technique will inform the developer that he will need to think about the method `setProxyCredentials(AuthScope,Credentials)` of the class `HttpState`. This hypothesis on the co-usage relationship between `setProxy` and `setProxyCredentials` methods is due to our analysis of various client programs of the HttpClient API, which found that both methods are always called together.

For instance, Figure 1 shows two code snippets of `HostConfiguration.setProxy` method[1]. In these code snippets, the `HttpState.setProxyCredentials` method is also invoked, after `setProxy`. These code snippets are from two different projects *MailBridege*[2] and *jUploader*[3]. In the first code snippet, the methods of interest are used in the constructor of `NetTools` class, while in the second code snippet they are used in a factory of `HttpClient` objects (in `HttpClientFactory.getHttpClient()`). The purposes of these client classes/methods are completely different as their names indicate. It is worth noting that in these client methods, as in others, the methods of interest (`setProxy` and `setProxyCredentials`) are used with other methods of HttpClient, such as `HttpClient.getParams` and `HttpClient.setParameter`. However, deeply analyzing a variety of client methods to `setProxy` method, we observed that its co-usage relationship with `setProxyCredentials` method remains consistent, but its co-usage relationships with other methods of HttpClient API are not.

[1]With the query *"setProxy & language = Java"*, the Ohloh code search engine returned more than 5 million results (the query was on 15th June 2014). Verifying all the code snippets found is practically impossible. However, most of the results that we were able to verify were not relevant to the method of interest (`HostConfiguration.setProxy`), but for other methods having the same name (*"setProxy"*).

[2]MailBridege is a HTTP tunnel for POP and SMTP access via a proxy.

[3]jUploader is a cross platform and cross-site photo uploader.

Actually, for a given HTTP client, once proxy settings are set (using `setProxy` method), the client program should consider updating the HTTP client's state, which contains all HTTP attributes that may persist from request to request (such as authentication credentials). Therefore, the client program should set the proxy credentials for the given authentication realm using `setProxyCredentials` method.

Despite the very well elaborated documentation of the HttpClient API, we had to spend a noticeable time to figure-out the above-mentioned information in this documentation. Indeed, reading the Java doc of the aforementioned classes and methods, and reviewing the HttpClient's code sample[4], we did not find any documentation regarding the co-usage relation between `HostConfiguration.setProxy` and `HttpState.setProxyCredentials`. Finally, reading the Authentication[5] section in the HttpClient's user guide, we found that the first paragraph "Server Authentication" and the fourth paragraph "Proxy Authentication" provide an *implicit* documentation of that co-usage relationship.

B. The Swing GroupLayout's Interface

To better illustrate the benefit of identifying consistent API usage patterns, we consider the class `GroupLayout`[6] in the Swing API [12]. `GroupLayout` is a LayoutManager that hierarchically groups components in order to position them in a Container. It is typically used by every client programs of the Swing API for managing layouts in JPanels. The (public) `GroupLayout`'s interface consists of 30 methods, and the class defines one constructor `GroupLayout(Container host)`. Reading the `GroupLayout`'s Java doc, the associated examples and the Java Swing tutorial, *"How to Use GroupLayout"*[7], we found that it is not trivial at all to identify the smallest subset of `GroupLayout`'s methods that are actually required for managing Swing GUI layouts. Indeed, the example provided with the `GroupLayout` documentation uses, in addition to the `GroupLayout`'s constructor, 6 methods of the `GroupLayout`'s interface. In the Java Swing tutorial, the provided example on `GroupLayout`[8] uses additional method (7 methods in total).

Analyzing a wide variety of client methods to the `GroupLayout` class, we found that a relatively small subset of the `GroupLayout`'s methods are always (consistently) used together to build layouts of Swing GUIs. Those methods are: the `GroupLayout`'s constructor, then `setHorizontalGroup` and `setVerticalGroup` methods. Hence, these methods represent a usage pattern of Swing API, that we refer to as *core usage pattern* of `GroupLayout`. Moreover, our analysis revealed that building a layout (using GroupLayout) cannot be complete without using either `createParallelGroup`, `createSequentialGroup`, or both methods, for specifying the type of layout's horizontal and vertical groups. In other words, the set {`createParallelGroup`,

[4]svn.apache.org/viewvc/httpcomponents/oac.hc3x/trunk/src/examples/

[5]hc.apache.org/httpclient-3.x/authentication.html

[6]http://docs.oracle.com/javase/7/docs/api/javax/swing/GroupLayout.html

[7]http://docs.oracle.com/javase/tutorial/uiswing/layout/group.html

[8]http://docs.oracle.com/javase/tutorial/uiswing/layout/groupExample.html

```
public class NetTools {
   private HttpClient client = null;
   ...
    public NetTools() {
      ...
         client.getHostConfiguration().setProxy( proxyHost, proxyPort );
         ...
         client.getState().setProxyCredentials( AuthScope.ANY, defaultcreds );
         ...
```

```
public class HttpClientFactory {
   public static HttpClient getHttpClient() {
      HttpClient client = new HttpClient(new MultiThreadedHttpConnectionManager());
      ...
         HostConfiguration hc = client.getHostConfiguration();
         hc.setProxy(proxyHost, proxyPort);
            ...
         client.getState().setProxyCredentials(scope, creds);
      ...
```

Figure 1. Code snippets of "`setProxy`" found using Ohloh code search engine.

`createSequentialGroup`} is, *partially* or *totally*, used with the core usage pattern of GroupLayout. We call these methods as *peripheral usage pattern* of `GroupLayout`.

In summary, our technique for mining API usage patterns can inform Swing users that layouts can be built using 5 methods: the core usage pattern (3 methods) and peripheral usage pattern (2 methods) of `GroupLayout`. Hence, developers can focus only on 5 methods of the `GroupLayout`'s interface (instead of 30 methods) for building the layouts of their Swing GUIs. Then, as needed, developers can then modify properties of their GUI layouts using other methods in `GroupLayout`.

III. Multi-Level Usage Pattern Analysis Approach

In this section, we detail our approach for detecting multi-level API usage patterns. Before detailing the used algorithm, we provide a deeper definition of multi-level usage patterns and their representation in our approach.

A. Multi-Level API Usage Patterns

As outlined earlier, we define an API usage pattern (UP) as a group (i.e., cluster) of methods of the API of interest that are co-used together by the API client programs. An UP includes only methods which are accessible from client programs (i.e., public methods of the API), and each UP represents an exclusive subset of the API's public methods.

Ideally, the co-usage relations between the UP's methods remain the same across *all possible* client methods of the UP. However, APIs are open applications, and it is unfeasible to analyze all their possible usage scenarios. Hence, we need a technique that can identify API usage patterns independently of the variability of features provided by the API's client programs and of the API's usage scenarios. Therefore, our technique for identifying API usage patterns should (1) capture out interference in co-usage relationships between the API's methods, and (2) isolate noises with respect to the degree of co-usage relationships in detected patterns.

To illustrate the representation of multi-level usage patterns, we use our example on `GroupLayout` class (Section II-B). In this example, we showed that the core usage pattern of `Grouplayout` consists of 3 methods, which are used together by all analyzed client methods of `Grouplayout`. We also outlined that the `Grouplayout` core usage pattern was always used with other methods of Swing. The issue here is to associate this usage pattern to other methods that are closely related to it and can enhance its informativeness, and to isolate it from other methods that can degrade its co-usage relationships and consistency. To address this issue, we incrementally cluster this core usage pattern with closely related methods, from the closest to the least close ones, so that the resulted multi-level usage pattern of `Grouplayout` will include the core and peripheral usage patterns of `Grouplayout` as shown in Figure 2. This incremental clustering provides valuable information: all client methods of the `Grouplayout` class, which invoke the `Grouplayout(Container)` constructor, utilize methods in cluster *L0* for `Grouplayout` initialization; most of these client methods create parallel groups using the `createParallelGroup` method; and the other client methods create sequential groups using the `createSequentialGroup` method, or create both parallel and sequential groups.

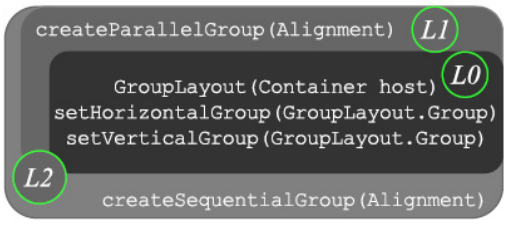

Figure 2. The cluster *L2* which represents the MLUP of class `GroupLayout`: *L0* represents the `GroupLayout`'s *core* usage pattern, then the cluster *L1*/*L2* includes *partially*/*totally* the `GroupLayout`'s *peripheral* usage pattern.

B. Approach Overview

Our approach takes as input the source code of the API to study and multiple client programs making use of this API. The output of our approach is a set of usage patterns as described in Section III-A.

The API usage patterns detection approach proceeds as follows. First, the API's and client programs source code is statically analyzed to extract the references between the methods of the client programs and the public methods of

the API. The static analysis is performed using the Eclipse Java Development Tools (JDT). Second, we compute usage vectors for the API public methods. Each public method in the API is characterized by a usage vector which encodes information about its client methods. Finally, we use cluster analysis to group the API methods that are most frequently co-used together by client methods.

C. Clustering Algorithm

Our clustering is based on the algorithm DBSCAN [15]. DBSCAN is a density based algorithm, i.e.; the clusters are formed by recognizing dense regions of points in the search space. The main idea behind it, is that each point to be clustered must have at least a minimum number of points in its neighborhood. This property of DBSCAN permits the clustering algorithm to leave out (not cluster) any point that is not located in a dense region of points in the search space. In other words, the algorithm clusters only relevant points and leaves out noisy points. This explains our choice of DBSCAN to detect API usage patterns. Indeed, not all public methods of the API are to be clustered because some are simply not co-used together nor with specific subsets of the API methods.

In our approach, each API public method is represented by a usage vector that has constant length l, that is the number of all client methods which use the API methods. Figure 3 shows that the API of interest is used by 7 client methods. And, these client methods use 5 methods of the API. Note that the client methods could belong to different client programs of the considered API. For an API method, m, an entry of 1 (or 0) in the i^{th} position of its usage vector, denotes that m is used (or not used) by the i^{th} client method corresponding to this position. Hence, summing the entries in the method's usage vector represents the number of its client methods. For instance, in Figure 3, the usage vector of *API.m1* shows that this API method is used by 4 client methods, which are *C1.m1*, *C1.m2*, *C2.m3* and *C3.m1*. We can see that these client methods use also the API methods *API.m2, API.m3* and *API.m4*, but do not use *API.m5*.

DBSCAN constructs clusters of API methods by grouping those that are close to each other (i.e., similar methods) and form a dense region. For this purpose, we define the Usage Similarity, $USim$ in Equation (1), between two API methods m_i and m_j, using the Jaccard similarity coefficient with regards to the client methods, Cl_mtd, of m_i and m_j. The rationale behind this is that two API methods are close to each other (short distance) if the corresponding methods share a large subset of common client methods.

$$USim(m_i, m_j) = \frac{|Cl\_mtd(m_i) \cap Cl\_mtd(m_j)|}{|Cl\_mtd(m_i) \cup Cl\_mtd(m_j)|} \quad (1)$$

Where $Cl\_mtd(m)$ is the set of client methods of the API method m. For example, the $USim$ between the API methods *API.m1* and *API.m2* in Figure 3 is $\frac{4}{5}$ –since these API methods have in total 5 client methods, and 4 of them are common for *API.m1* and *API.m2*. The distance between two methods m_i and m_j is then computed as $Dist = 1 - USim(m_i, m_j)$.

DBSCAN needs two parameters to perform the clustering. The first parameter is the minimum number of methods in a cluster. We set it at two, so that a usage pattern must include at least two methods of the studied API. The second parameter, epsilon, is the maximum distance within which two methods can be considered as neighbor, each to other. In other words, epsilon value control the minimal density that a clustered region can have. The shorter is the distance between the methods within a cluster the more dense is the cluster. As we will show in the next section, we will use different values for epsilon for identifying multi-level usage patterns.

	C1.m1	C1.m2	C2.m1	C2.m2	C2.m3	C3.m1	C3.m2
API.m1	1	1	0	0	1	1	0
API.m2	1	1	1	0	1	1	0
API.m3	0	1	1	0	1	0	0
API.m4	1	1	1	0	1	1	0
API.m5	0	0	0	1	0	0	1

Figure 3. The usage vector representation of five API methods with seven client methods.

```
HDBSCAN(DataSet, maxEpsilon, MinPts,
    espilonStep){
  epsilon <- 0
  while(epsilon < maxEpsilon)
       DBSCAN(DataSet, epsilon, MinPts)
       clusters <- DBSCAN.clusters
       noisyPoints <- DBSCAN.noisyPoints
       compositePoints <- constructPoints(
           clusters)
       DataSet <- noisyPoints + compositePoints
       epsilon <- epsilon + epsilonStep
}

constructPoints(clusters){
  for each C in clusters
       compositePoint <- OR(all points of C)
}
```

Figure 4. Hierarchical DBSCAN algorithm.

D. Incremental Clustering

In DBSCAN, the value of the epsilon parameter influences greatly the resulting clusters. Indeed, in our approach, a value of 0 for epsilon, means that each cluster must contain only API methods that are completely similar (i.e., distance among methods belonging to the same cluster must be 0). Relaxing the epsilon parameter will relax the constraint on the requested density within clusters.

On the one hand, if we set epsilon at fixed small value, such as epsilon $= 0$, this will produce usage patterns that are very dense. Yet, resulted usage patterns will not capture out interference in co-usage relationships between the API's methods: a usage pattern will include only methods that are all always co-used together. On the other hand, fixing epsilon to relatively large value, such as epsilon $= 0.6$, DBSCAN will produce usage patters that include some noises. And, for a given usage pattern, it will not be easy to distinguish between dense subsets from subsets that capture out interference in co-usage relationships.

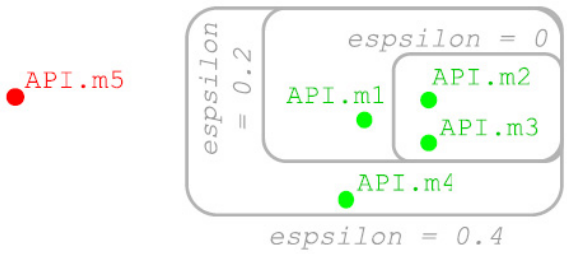

Figure 5. Resulting clusters of applying the incremental algorithm to API methods of Figure 3.

In our approach, we decided to build the clusters incrementally by relaxing the epsilon parameter, step by step. Figure 4 shows the pseudo-code of the incremental clustering. First, we construct a dataset containing all the API methods and cluster them using DBSCAN algorithm with epsilon value of 0. This results in clusters of API methods that are always used together, and multiple noisy API methods left out. At the end of this run, for each produced cluster we aggregate the usage vectors of its methods using the logical disjunction in one usage vector. Then, a new dataset is formed which includes the aggregated usage vectors and the usage vectors of noisy methods from the first run. This dataset is fed back to the DBSCAN algorithm for clustering, but with a slightly higher value of epsilon, that is epsilon = $0+\delta$. This procedure is repeated in each step corresponding to an epsilon value of α. And, the clustering process is stopped when epsilon reaches a maximum value, β, given as parameter.

For example, Figure 5 shows that result of our incremental clustering for the API methods in Figure 3. In this example, the initial dataset contains 5 methods, *API.m1, ..., API.m5*, the epsilon parameter is incremented in each step by $\delta = 0.2$, and the epsilon maximum value was set to $\beta = 0.5$. Figure 5 shows that the algorithm will produce one multi-level usage pattern that contains in total 4 methods (*API.m1, ..., API.m4*), and *API.m5* is left out as a noisy method. The produced multi-level usage pattern involves 3 levels of density. The first level, which is the most dense one, is clustered at epsilon $= 0$, and it includes only *API.m2* and *API.m4*. These two methods represents the core of the identified usage pattern since they are always co-used together (i.e., they have a perfect usage similarity). The second level, is clustered at epsilon $= 0.2$, and includes *API.m1* in addition to *API.m2* and *API.m4*. This method, *API.m1*, is included in this level since its distance from both *API.m2* and *API.m4* is smaller than 0.2. Finally, the third level, which is the less dense one, is clustered at epsilon $= 0.4$. This level includes, in addition to the methods of the first and second level, the method *API.m3*. At the end, the method *API.m5* is left out as a noisy method since its distance from any clustered method is larger than epsilon maximum value, which is 0.5. This multi-level usage pattern can be interpreted as that the pattern's core, which includes *API.m2* and *API.m4*, can have 2 interference usage scenarios: (1) the core methods can be co-used, most frequently, with *API.m1*, which shares 4/5 common client methods; (2) the core methods can be co-used with *API.m3*, which shares 3/5 common client methods.

IV. Evaluation

The objective of our study is to evaluate whether our technique can detect API usage patterns which are enough cohesive to provide valuable information that can help in learning and using APIs, and which are generalizable independently of the API usage contexts. We formulated the research questions of our study as follows:

- **RQ1:** to which extent the detected usage pattern are cohesive?
- **RQ2:** to which extent the detected usage pattern could be generalized to other "new" client programs, that are not considered in the mining process?

A. Data

We evaluate our technique through the usage of four widely used APIs: HttpClient, Java Security, Swing and AWT (Table IV-A). To perform our study, we selected 22 client programs for the Swing and AWT APIs, and 12 client programs for the HttpClient and Java Security APIs (see Table IV-A).

B. Comparative Evaluation

To address our research questions we opted for a comparative evaluation, we compared our technique for mining multi-level API usage patterns (MLUP) to the most similar approche MAPO as configured in [2]. To mine API usage patterns, MAPO clusters frequent API method call sequences extracted from client programs, then use the BIDE [16] algorithm to mine closed sequential patterns from the preprocessed method-call sequences: i.e., to capture groups of API's method that are frequently used together. This comparison allow us to better position our approach and characterize the obtained results. For MLUP we fixed the `maxEpsilon` value to 0.35. Obtained results are not computed for the core and the peripheral patterns separately, but rather for the usage pattern as a whole.

C. Metrics and Experimental Setup

The following describes the process of our experiments for a given API of the case study. The setup settings described in this section were applied for the two compared approaches MLUP and MAPO.

1) Pattern Co-Usage Relationships: To address our first research question **(RQ1)**, we need to evaluate whether detected usage patterns are enough cohesive to exhibit informative co-usage relationships between the API methods. To measure the cohesion of usage patterns, we use the Service Usage Cohesion Metric (SIUC), taken from Perepletchikov *et al.* [17]. SIUC was recently used in [18] to measure the usage cohesion of interfaces in Java OO applications, where, like our definition of a usage pattern, an interface is defined as a group of method

Table I
Selected APIs for the Case Study

API	Description
Java Security	Provides features to improve security of Java applications
HttpClient	Implements standards for accessing resources via HTTP
Swing	An API providing a graphical user interface (GUI) for Java programs
AWT	An API for providing a platform dependent graphical user interface (GUI) for a Java program

Table II
CLIENT PROGRAMS USED IN OUR CASE-STUDY

APIs	Client programs	Description
Swing & AWT	G4P (GUI for processing)	A library that provides a rich collection of 2D GUI controls
	Valkyrie RCP	A Spring port of the current Spring Rich Client codebase.
	GLIPS Graffiti editor	A cross-platform SVG graphics editor
	Mogwai Java Tools	Java 2D and 3D visual entity relationship design and modeling (ERD,SQL)
	Violet	An UML editor
	Mobile Atlas Creator	This application creates off-line raster maps
	Metawidget	A smart widget Building User Interfaces for domain objects
	Art-of-Illusion	A 3D modelling and rendering studio
	VASSAL	An engine for building and playing human-vs-human games
	Neuroph	A lightweight Java Neural Network Framework
	WoPeD	A Java-based graphical workflow process editor, simulator and analyzer
	jEdit	A mature programmer's text editor
	Spring-RCP	Provide a way to build highly-configurable, GUI-standards
	GanttProject core	An application for project management and scheduling
	Pert	The PERT plugin for GanttProject
	Htmlpdf	The html and pdf export plugin for GanttProject
	Msproject	MS-Project import/export plugin for GanttProject
	swingx	Contains extensions to the Swing GUI toolkit
	JHotDraw	A Java GUI framework for technical and structured Graphics
	RapidMiner	An integrated environment for machine learning and data mining
	Sweet Home 3D	An interior design application
	LaTeXDraw	Is a graphical drawing editor for LaTeX
HttpClient & Java Security	Jakarta Cactus	A simple test framework for unit testing server-side java code
	Apache JMeter	A project that can be used as a load testing and measure performance tool
	Heritrix	A web crawler
	HtmlUnit	A GUI-Less browser for Java programs
	OpenLaszlo	An open source platform for the development and delivery of rich Internet applications
	Mule	A lightweight enterprise service bus (ESB) and integration framework
	RSSOwl	An aggregator for RSS and Atom News feeds.
	Apache Jackrabbit	Is an open source content repository for the Java platform.
	Axis2	A core engine for Web services.
	RESTEasy	A JBoss project that provides various frameworks to build RESTful Web Services
	WildFly	An application server
	WSO2 Carbon	An SOA middleware platform

declarations. In the following, we refer to the SIUC metric as Pattern Usage Cohesion (PUC). PUC takes its value in the range [0..1]. The larger the value of PUC is, the better the usage cohesion. PUC states that a usage pattern has an ideal usage cohesion (PUC = 1) if all the pattern's methods are actually always used together. The PUC for a given usage pattern *p* is defined as follows:

$$PUC(p) = \frac{\sum_{cm} ratio\_used\_mtds(p, cm)}{|CM(p)|} \quad (2)$$

Where *cm* denotes a client method of the pattern *p*; *ratio_used_mtds(p, c)* is the ratio of methods which belong to the usage pattern *p* and are used by the client method *cm*; and $CM(p)$ is the set of all client methods of the methods in *p*.

To answer our first research question **(RQ1)**, we apply our technique and MAPO for all the selected APIs of the case study, using the APIs client programs described in Table IV-A. Then we compare the cohesion of the detected usage patterns through the two techniques, using the PUC metric.

2) Pattern Generalization: The detection of usage patterns for an API depends on the used set of API's client programs (training client programs). Hence, to address our second research question **RQ2**, we need to evaluate whether the detected API's usage patterns will have similar usage cohesion in the context of new client programs of the API (validation client programs). Our hypothesis is that: *detected usage patterns for an API are said "generalizable" if they remain characterized by a high usage cohesion degree in the contexts of various API client programs.* This is regardless of the natures and features of those client programs, and of whether those programs were used or not for detecting the API's usage patterns. Such generalizable usage patterns can contribute to learn common ways of using the API of interest.

To evaluate the generalizability of detected patterns, we perform leave-one-out cross-validations for all the selected APIs of the case study, using the APIs client programs described in Table IV-A. Let N represents the number of used client programs for the considered API (e.g., N = 22 for Swing), we perform N runs of the two compared techniques (MLUP and MAPO) on the API. Each run uses N-1 client programs as training client programs for detecting usage patterns, and leaves away one of the APIs client programs as validation client programs. The results are sorted in N runs, where each run has its associated usage patterns, and its corresponding training and validation client programs.

Then, we address our second question **(RQ2)** in two steps, as follows. In the first step, we evaluate the cohesion of the detected usage patterns (as measured by PUC) in the contexts of validation sets. However, in a given run, it is possible that some detected usage patterns involve only methods that are not used at all in the validation client programs. Therefore, to evaluate the generalizability of detected patterns in a run, we consider only patterns which contain at least one method that is actually used by the run's validation client programs. We call such patterns as the eligible patterns for the validation client programs. For an eligible pattern, if only a small subset of its methods are used by the validation client programs, while the other methods are not used, the pattern will have a low usage cohesion. As a consequence, it will be evaluated as "not generalizable". At the end of this step, we compare between the cohesion results obtained with MLUP and MAPO.

In the second step, for each run we evaluate the consistency of the detected usage patterns between the training client programs and the validation client programs. A pattern is said consistent if the co-usage relationships between the pattern's methods in the context of the training client programs remain the same (or very similar) in the context of the validation client programs. We define the consistency of a usage pattern as:

$$Consistency(p) = 1 - |PUC_T(p) - PUC_V(p)| \quad (3)$$

Where PUC_T and PUC_V are the usage cohesion values of the pattern p in the training client programs context and validation client programs context, respectively. This metric takes its value in the range [0..1]. A value close to 1 indicates that the co-usage relationships between the pattern's methods remain the same between training client programs and the *new* validation client program, while a value close to 0 indicates a dissimilar behavior of the pattern between the two sets of clients. This metric allows us to see whether between changing contexts good patterns remain good ones and bad patterns remain bad ones.

V. Results Analysis

Before addressing our research questions, we analyzed the dependencies between the client programs in Table IV-A and the selected APIs. We applied the two compared techniques for detecting usage patterns of selected APIs. Table V summarizes the results of this phase, which shows that, for all studied APIs, our technique (MLUP) has been able to detect more usage patterns than MAPO. And, MLUP usage patterns are, overall, larger than MAPO ones. In the following, we investigate the results of our experiments explained in Section IV to address our research questions **RQ1** and **RQ2**.

A. Patterns Cohesion (RQ1)

To answer our research question RQ1, we analyze the average and the distribution of usage cohesion values for all detected usage patterns per studied API, using the two compared approaches. Table V-A clearly show that, in average, MLUP outperforms MAPO for detecting cohesive usage patterns. Moreover the Wilcoxon rank sum test statically confirms these statements. The usage cohesion values obtained for MLUP reflect very strong co-usage relationships between the pattern's methods for all studied APIs. Indeed, the average usage cohesion values for MLUP are between 90% and 96%, whereas MAPO's average values are between 55% and 71% .

Table III
Overview on the number of covered/analyzed methods and the number of detected usage patterns per API

		MLUP				MAPO			
			UP's size				UP's size		
API	**cov. mtd**	**UPs**	**Avg**	**Min**	**Max**	**UPs**	**Avg**	**Min**	**Max**
Java Security	125	12	2.8	2	4	4	2.0	2	2
HttpClient	343	20	2.7	2	4	12	2.5	2	4
Swing	1618	102	3.9	2	7	69	2.0	2	2
AWT	1019	75	4.3	2	9	50	3.1	2	6

Table IV
Average Cohesion of identified API usage patterns, for MLUP and MAPO.

API	**MLUP**	**MAPO**
Java Security	0.90	0.71
HttpClient	0.96	0.55
Swing	0.94	0.61
AWT	0.94	0.60

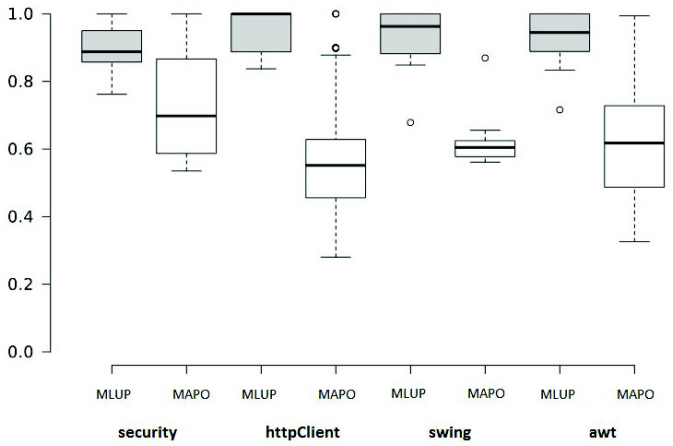

Figure 6. Cohesion values of identified API usage patterns, for MLUP (gray boxes) and MAPO (white boxes).

The distribution of usage cohesion values for all detected usage patterns in Figure 6 confirms the above-mentioned finding. Indeed, in the worst case for MLUP and the best case for MAPO, the case of Java Security API, the median usage cohesion with MLUP and MAPO are respectively around 90% and 71%. For the other studied APIs, the medians and lower quartiles remain larger than 90% for MLUP, whereas with MAPO the medians and upper quartiles persist under 70%.

In summary, the co-usage relationships between the methods of every usage pattern detected with MLUP are comparatively strong, where at least 70%, and upto 100%, of the pattern's methods are co-used together.

B. Patterns Generalization (RQ2)

The cross-validation allow us to observe the generalizability of the detected patterns on two levels. First we inspect the co-usage relationships of detected patterns in the context of potential new client programs. Then we characterize the usage cohesion deterioration between the training and validation

Table V
STATISTICS ON THE COHESION OF IDENTIFIED API USAGE PATTERNS FOR MLUP AND MAPO , IN VALIDATION CLIENTS

	MLUP			MAPO		
API	**Avg**	**StdDev**	**Max**	**Avg**	**StdDev**	**Max**
Java Security	0.85	0.12	1.00	0.67	0.12	1.00
HttpClient	0.79	0.16	1.00	0.45	0.08	0.58
Swing	0.85	0.09	1.00	0.58	0.03	0.64
AWT	0.84	0.06	0.95	0.39	0.09	0.67

Table VI
STATISTICS ON THE CONSISTENCY OF IDENTIFIED API USAGE PATTERNS FOR MLUP AND MAPO

	MLUP			MAPO		
API	**Avg**	**StdDev**	**Max**	**Avg**	**StdDev**	**Max**
Java Security	0.86	0.09	0.98	0.87	0.06	0.96
HttpClient	0.77	0.09	0.89	0.86	0.03	0.90
Swing	0.83	0.05	0.92	0.91	0.02	0.95
AWT	0.84	0.04	0.94	0.77	0.03	0.88

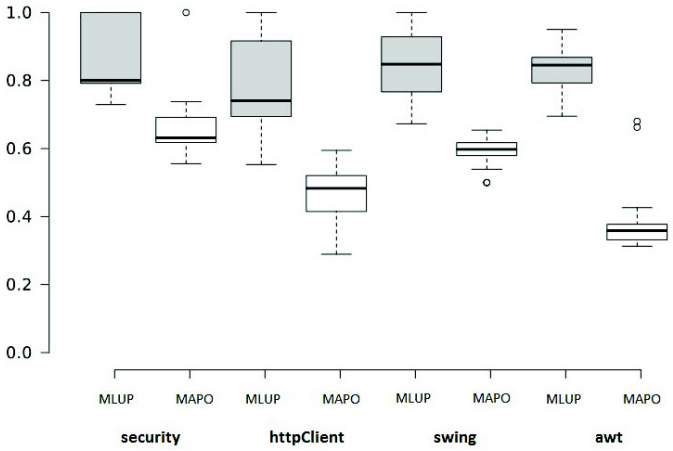

Figure 7. Cohesion values of identified API usage patterns, for MLUP and MAPO in the contexts of validation clients

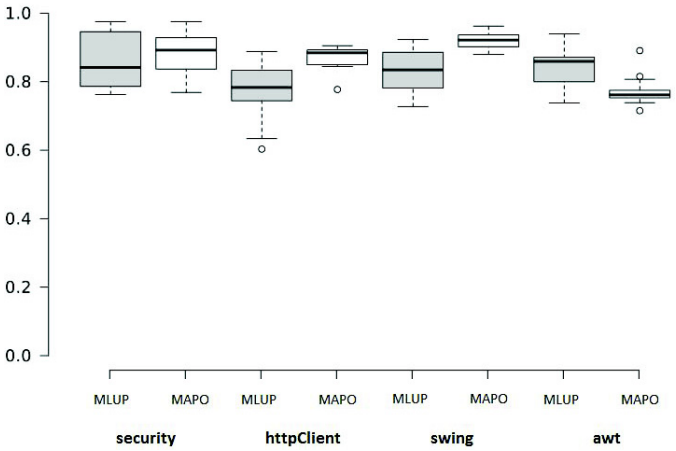

Figure 8. Consistency values of identified API usage patterns for MLUP and MAPO, across multiple validation clients.

client programs. In both levels, we first analyze the average value of corresponding measurements collected from all cross-validation runs (Table V-B1 and Table V-B2), and then we analyze the distribution of collected values using median boxplots (Figure 7 and Figure 8).

1) Patterns Validation Cohesion: Table V-B1 summarizes, for both MLUP and MAPO, the usage cohesion of detected usage patterns in the contexts of validation client programs. For MLUP we notice that the average values remain high (around 85%), but with a slight degradation in the case of HttpClient API where the average usage cohesion is 79%. We also notice that the standard deviation values are very low. This reflects that, overall, the detected patterns using MLUP had always very good usage cohesion in the context of validation client programs. As for MAPO, the average usage cohesion values are significantly lower.

In Figure 7, although patterns cohesion values were degraded for both MLUP and MAPO, as compared to the patterns cohesion values in Figure 6, we observe that the median values for MLUP remain high. We also notice that the degradation of cohesion values is much more visible for MAPO. Precisely, for MLUP, in the worst case (HttpClient API) the median usage cohesion for detected patterns is around 75%. For the other studied APIs, the medians and lower quartiles remain larger than 80%.

In summary, compared to usage patterns that are detected by MAPO, usage patterns detected with our technique remain highly cohesive across various client programs of the API of interest. This shows that MLUP detected usage patterns retain their informative criteria independently of the API usage scenarios. Examples of such API usage patterns are the patterns that we discussed in Sections II-A and II-B. These results show that API usage patterns detected with our technique can be used to enhance the API documentation with co-usage relationships with high confidence without a need to consider all possible usage contexts (client programs) of the API of interest.

2) Patterns Consistency: As it can be seen in Table V-B2, the results show that, using both MLUP and MAPO, the identified usage patterns for the four studied APIs are overall characterized with high consistency. Indeed, for both MLUP and MAPO, the average consistency values of detected patterns across multiple validation client programs are around 80%. The table also shows that maximum consistency values for all studied APIs are very close to the ideal value, which is 1, and the standard deviation values are very small. We consider as example the case of AWT API, where 22 leave-one-out cross validations have been made, using 22 client programs. In this case the results show that, for any AWT's detected pattern, on average 84% (for MLUP) and 77% (for MAPO) of co-usage relationships between pattern methods remain similar across 22 client programs of AWT.

The boxplots in Figure 8 give more information on the consistency of detected usage patterns in each run of the compared techniques. Overall, the boxplots in Figure 8 show that, for detected patterns with MLUP, the median and lower quartile values are almost equal for 3 APIs (Java Security, Swing, and AWT), and they are around 85% and 80%, respectively. Hence, for these 3 APIs, we observe that almost all detected usage patterns (precisely 75% of detected usage

patterns, according to the box lower quartile) are characterized with a high consistency, where the pattern's consistency value is greater than 80%. In comparison with MAPO results for these 3 APIs, we observe that the consistency of usage patterns detected with MLUP is comparable to that of MAPO, with a very small delta in favor of MAPO in some cases. However, recalling that the cohesion scores for MLUP were much higher than those for MAPO, in the context of both validation sets (Figure 7) and training sets (Figure 6), we believe that this degradation of patterns consistency for our technique MLUP is actually acceptable.

In summary, the consistency metric reflects only the behavioral similarity of detected patterns with reference to the training and validation client programs, regardless of the cohesion quality. Both techniques are equivalent in terms of consistency, while MLUP remain much better in terms of cohesion quality of detected patterns.

VI. Discussion

We applied our approach to four APIs and detected usage patterns that are informative and which could help to enhance the API's documentation. The evaluation of our approach took into account the generalization to other client programs of the same API and showed that the usage pattern of an API remain informative for other clients. Although, we selected four APIs of different domains, the approach may not be generalized to all APIs because they may be of different natures.

The API usage pattern detected by our technique have the valuable property of generalizability. An API pattern detected using a certain set of client programs of the API, as input to our approach, is very likely to be evenly detected using a different set of client programs of the API of interest. This aspect of our approach also ensures that the detected patterns are independent of specific usage scenarios. The generalizability of the patterns suggests that they are used by different client programs involving various features. From this perspective, our approach is meant for enriching the API documentation and facilitating the development tasks when using new APIs.

One of the key contributions of our work is the adaptation of DBSCAN algorithm for mining API usage patterns. We have opted for this technique rather than for classic hierarchical clustering techniques since DBSCAN has a notion of noise, and is robust to outliers. Indeed, when mining API usage patterns, it is very probable to find many API methods that are not used jointly with specific set of methods. Such methods are utility methods that can be used with distinct subsets of the API methods. These methods are considered as noise and should not be clustered.

The efficacy of our approach relies on the coverage of the API methods, provided by the used client programs for mining the API usage patterns. This coverage has also to be redundant, i.e., an API method should be covered (used by) several client methods from different client programs. This limitation is shared by all existing work around making use of multiple API clients.

Finally, the application of our technique to detect multi-level API usage patterns requires the setting of thresholds that may impact its output. For instance, the epsilon parameter in the clustering algorithm controls the *co-usage strength* of the detected patterns. A small value leads to highly cohesive clusters which means that the detected patterns are very informative. Hence, decreasing the value of this parameter would result in an improvement in cohesion of detected patterns. However, in this case the number and the consistency of detected patterns could decrease because the highly cohesive detected patterns may not be shared by a large number of clients. Therefore, there is a tradeoff between the co-usage relationship of our detected patterns and their generalization to a large number of clients. Based on our initial experience we set the `maxEpsilon` value to 0.35 which reflects a similarity of 0.65 but this value still need more investigation

VII. Related Work

Recently, different aspects around APIs usage have gained considerable attentions. Existing contributions can be organized in the following categories according to the purpose of their proposed techniques: code completion, API usage example recommendation, API usage visualization, exploration of API usage obstacles, and mining API usage patterns.

Code Completion: Enhancing current completion systems to work more effectively with large APIs have been investigated in [9], [8], [19]. This body of work makes use of database of API usage recommendation, type hierarchy, context filtering and API methods functional roles for improving the performance of API method call completion. Recently, Asaduzzaman *et al.* [20] proposed a context sensitive code completion technique that uses, in addition to the aforementioned information, the context of the target method call.

API Usage Example Recommendation: A similar body of work is interested in example recommendation of API usage [21], [22], [23], [24]. Existing contributions can be organized in two groups: IDE-based recommendation systems and JavaDoc-based recommendation systems. These contributions tried to instruments API documentation with usage examples based on a static slicing, clustering and pattern abstraction.

API Usage Visualization: Other contributions tried to enhance understanding API usage through explorative and interactive methods [25], [26]. This body of work described multi-dimensional exploration of API usage. The explored dimensions are related to the hierarchical organization of projects and APIs, metrics of API usage and API domains.

Exploration of API Usage Obstacles: From another perspective, the work in [27], [28] explored API usage obstacles through analyzing developers questions in Q&A website. This allows API designers to understand the problems faced while using their API, and to make corresponding improvements.

Mining API Usage Patterns: The most related contributions to ours are those interested in mining API usage patterns [29], [1], [2], [30]. These contributions adopted different categories of API usage patterns, different techniques for inferring patterns and different ways to assess patterns correctness and usefulness. The most prominent categories are temporal ([1]), unordered ([30]) and sequential ([29], [2]) usage patterns. These categories were assessed through consistency, coverage and succinctness of the mined usage patterns. The authors in [29] also compared their approach (UP-Miner) to MAPO

[2]. Results show that UP-Miner reduces number of redundant patterns and detects more *unpopular* patterns, as compared to MAPO. As UP-Miner and MAPO are comparable and both are based on the BIDE algorithm to mine closed sequential patterns, and as we are interested in detecting *popular* (i.e., generalize-able) patterns, we chose to compare our approach to MAPO. Although MAPO mines sequential patterns, the sequential ordering of methods within usage patterns has not any impact on the comparison criteria (i.e., usage cohesion and consistency) adopted in our study.

The main limitations of the above-mentioned contributions are the inability to detect common/consistent API usage patterns, inability to capture out less common (interfering) API usage scenarios, and the context dependency. Our approach tries to circumvent theses limitations. First, we make use of multiple clients of the studied API which guarantees that usage patterns are genuine and recovered from the actual use of the API, and they are generalize-able for new API client programs. Second, the hierarchical construction of our usage patterns enables the identification of core usage patterns, as well as peripheral usage patterns independently from usage context. Finally, our approach detects consistent usage patterns that are used almost in the same way by API client programs.

VIII. Conclusion

We developed a technique that identifies multi-level API usage patterns. We detect groups of API methods that are highly cohesive in terms of usage by client programs. We analyzed four APIs along with up to 22 client programs per API. Our approach detected API usage patterns that are generalizable for a wide variety of API client programs. The detected patterns are constructed in a hierarchical manner and are useful to enrich the API's documentation.

Our approach is based on static analysis of the API and its client program. A more precise analysis using dynamic analysis could improve the results. In the near future, we plan to evaluate our approach using a large dataset of APIs and to evaluate empirically the quality of detected usage patterns from developers perspective.

References

[1] G. Uddin, B. Dagenais, and M. P. Robillard, "Temporal analysis of api usage concepts," in *International Conference on Software Engineering*, 2012, pp. 804–814.

[2] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *European Conference on Object-Oriented Programming*, 2009, pp. 318–343.

[3] U. Dekel and J. D. Herbsleb, "Improving api documentation usability with knowledge pushing," in *International Conference on Software Engineering*, 2009, pp. 320–330.

[4] D. Schreck, V. Dallmeier, and T. Zimmermann, "How documentation evolves over time," in *international workshop on Principles of software evolution*, 2007, pp. 4–10.

[5] M. P. Robillard, "What makes apis hard to learn? answers from developers," *IEEE Software*, vol. 26, no. 6, pp. 27–34, 2009.

[6] "Ohloh code search engine." [Online]. Available: https://code.ohloh.net/

[7] R. Holmes, R. Walker, and G. Murphy, "Approximate structural context matching: An approach to recommend relevant examples," *Software Engineering, IEEE Transactions on*, vol. 32, no. 12, pp. 952–970, 2006.

[8] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2009, pp. 213–222.

[9] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Graph-based pattern-oriented, context-sensitive source code completion," in *International Conference on Software Engineering*, 2012, pp. 69–79.

[10] "The jakarta commons httpclient component." [Online]. Available: http://hc.apache.org/httpclient-3.x/

[11] "The java.security package and all its subpackages (5 public packages in total)." [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/security/package-summary.html

[12] "The swing api (18 public packages in total)." [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/swing/

[13] "The java.awt package and all its subpackages (12 public packages in total)." [Online]. Available: http://docs.oracle.com/javase/7/docs/api/

[14] T. Xie and J. Pei, "Mapo: Mining api usages from open source repositories," in *International workshop on Mining software repositories*, 2006, pp. 54–57.

[15] M. Ester, H. peter Kriegel, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *International Conference on Knowledge Discovery and Data Mining*, 1996, pp. 226–231.

[16] J. Wang and J. Han, "Bide: Efficient mining of frequent closed sequences," in *International Conference on Data Engineering*, 2004, pp. 79–90.

[17] M. Perepletchikov, C. Ryan, and Z. Tari, "The impact of service cohesion on the analyzability of service-oriented software," *Services Computing, IEEE Transactions on*, vol. 3, no. 2, pp. 89–103, 2010.

[18] H. Abdeen, H. Sahraoui, and O. Shata, "How we design interfaces, and how to assess it," in *International Conference on Software Maintenance*, 2013, pp. 80–89.

[19] D. Hou and D. M. Pletcher, "An evaluation of the strategies of sorting, filtering, and grouping api methods for code completion," in *International Conference on Software Maintenance*, 2011, pp. 233–242.

[20] M. Asaduzzaman, C. K. Roy, K. Schneider, and D. Hou, "Cscc: Simple, efficient, context sensitive code completion," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 71–80.

[21] L. Wang, L. Fang, L. Wang, G. Li, B. Xie, and F. Yang, "Apiexample: An effective web search based usage example recommendation system for java apis," in *International Conference on Automated Software Engineering*, 2011, pp. 592–595.

[22] E. Duala-Ekoko and M. P. Robillard, "Using structure-based recommendations to facilitate discoverability in apis," in *European Conference on Object-oriented Programming*, 2011, pp. 79–104.

[23] R. P. L. Buse and W. Weimer, "Synthesizing api usage examples," in *International Conference on Software Engineering*, 2012, pp. 782–792.

[24] J. E. Montandon, H. Borges, D. Felix, and M. T. Valente, "Documenting apis with examples: Lessons learned with the apiminer platform," in *Working Conference on Reverse Engineering*, 2013, pp. 401–408.

[25] E. Moritz, M. Linares-Vasquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers, "Export: Detecting and visualizing api usages in large source code repositories," in *Automated Software Engineering*, 2013, pp. 646–651.

[26] C. De Roover, R. Lammel, and E. Pek, "Multi-dimensional exploration of api usage," in *International Conference on Program Comprehension*, 2013, pp. 152–161.

[27] D. Hou and L. Li, "Obstacles in using frameworks and apis: an exploratory study of programmers' newsgroup discussions," in *International Conference on Program Comprehension*, 2011, pp. 91–100.

[28] W. Wang and M. W. Godfrey, "Detecting api usage obstacles: A study of ios and android developer questions," in *Working Conference on Mining Software Repositories*, 2013, pp. 61–64.

[29] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage api usage patterns from source code," in *Working Conference on Mining Software Repositories*, 2013, pp. 319–328.

[30] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 306–315.