Temporal Analysis of API Usage Concepts

Gias Uddin, Barthélémy Dagenais, and Martin P. Robillard
School of Computer Science
McGill University
Montréal, QC, Canada
{*gias, bart, martin*}@*cs.mgill.ca*

***Abstract*—Software reuse through Application Programming Interfaces (APIs) is an integral part of software development. The functionality offered by an API is not always accessed uniformly throughout the lifetime of a client program. We propose *Temporal API Usage Pattern Mining* to detect API usage patterns in terms of their time of introduction into client programs. We detect concepts as distinct groups of API functionality from the change history of a client program. We locate those concepts in the client change history and detect temporal usage patterns, where a pattern contains a set of concepts that were added into the client program in a specific temporal order. We investigated the properties of temporal API usage patterns through a multiple-case study of three APIs and their use in up to 19 client software projects. Our technique was able to detect a number of valuable patterns in two out of three of the APIs investigated. Further investigation showed some patterns to be relatively consistent between clients, produced by multiple developers, and not trivially derivable from program structure or API documentation.**

***Keywords*-API Usage; API Usability; Usage Pattern; Software Reuse; Mining Software Repositories**

I. Introduction

Software systems often reuse functionality provided by libraries and frameworks. The client program reuses functionality through Application Programming Interface (API). Despite advances in API documentation and assisting technologies [1, 8, 22], large APIs are still hard to learn [18]. A major challenge for API users is to discover the subset of the API that can help complete a task. For large APIs, there typically exists an overwhelming number of ways to combine different API elements. Hence, it can be particularly helpful to identify common *usage patterns* for the API.

Although many approaches have been proposed to detect common ways to use an API (see Section VII), existing technology does not provide guidance about *when* patterns are relevant in the life-cycle of clients, or whether there exist temporal relations *between* different API functionalities.

We propose *Temporal API Usage Pattern Mining*. Our notion of temporal API usage is founded on the observation that references to *cohesive subsets of an API are often introduced in client programs in a specific order*. We base our pattern detection strategy on the analysis of the introduction of references to API elements in the history of client programs. Our approach extracts client changes related to an API and uses a clustering technique to produce *concepts*, i.e., groups of API elements (typically methods) that were added together to implement a functionality. We then analyze the concepts in terms of the time they were introduced in a client, thus recovering temporal usage patterns.

Knowledge of temporal API usage patterns can inform developers of the next possible steps in using an API. The patterns can also be used to improve developer learning resources, e.g., by ensuring that tutorials cover each concept in order. Temporal patterns would thus add a new dimension to existing API recommendation systems, such as API Explorer [9] or Intelligent Code Completion Systems [4].

We investigated the potential value of this novel approach through a collective case study on the usage of three APIs of diverse nature (HttpClient, java.security, and java.util) in a total of 19 open source client programs. For HttpClient and java.security, we found that even if different client programs did not provide the same features, in many cases they followed a common set of usage patterns. For example, in the case of the HttpClient API, we detected that client programs that implemented cookie support also checked for the configuration of the host machine where cookie support was enabled (to determine whether the host is responding accordingly). Implementing these two concepts required the instantiation of three different HttpClient types, calls to five different methods and access to one field. None of these methods are structurally dependent on each other (i.e., the invocation of one method does not strictly require that the other methods be invoked before). The patterns we detect include not only these temporal relations between concepts, but also the list of API elements involved in each concept. We compared our detected concepts against the itemsets generated by frequent itemset mining. We found that the concepts detected by our technique are more informative (e.g., the groups of entities in a concept are more cohesive) and contain significantly fewer false positives than the baseline technique. We observed that the patterns generated are generally consistent across clients and can uncover interesting temporal dependencies among concepts. Our case study also shows how the detected patterns could have helped improve the documentation of the APIs.

A brief overview of a preliminary version of this approach has been showcased in a 4-page short paper [19]. This current paper provides the first complete report on our new, fully-implemented technique which includes conceptually-significant improvements to the approach, and a detailed analysis of a multiple-case study that provides evidence of the value of the approach.

II. Temporal API Usage

The idea of temporal API usage patterns is best described through an example. While initializing an HTTP connection using the HttpClient API,[1] a developer might first add a call to the `HttpClient` constructor and to the methods of the class `UserNamePasswordCredentials`. This API usage pattern establishes a basic interaction point with the HTTP server. Once this code is working, the developer may want to assess the connection status with the server by calling the `HttpMethodBase.getStatusText(...)` and comparing the responses with the API fields such as `HttpStatus.SC_OK`. In later development, the developer may call the methods of `GetMethod` to retrieve contents from the server. In both cases, a connection to the HTTP server should be established first.

For an API and a client, we detect a set of temporal patterns, where each pattern contains an ordered sequence of concepts that were implemented together in specific temporal order. A concept embodies an API functionality (e.g., establishing an HTTP connection) using some API elements. A temporal pattern P can be specified as follows:

$$P = \{m_1, m_2\} \rightarrow \{m_3, m_4\}$$

This pattern indicates that methods m_1 and m_2 formed a concept and this concept was introduced before the concept containing the methods m_3 and m_4. The sets of concepts inside a temporal pattern is ordered. A client program may exhibit more than one temporal pattern.

III. Analysis Framework

In this section, we describe our analysis framework to generate temporal API usage patterns. The input to our approach is the source history of a client program that uses the API of interest and the output is a set of temporal usage patterns as described above. Our approach involves four steps, each supported by our infrastructure.

1) **Processing change history.** We convert the unstructured textual differences between files versions in the client system into a database of transactions where all additions of references to methods and fields of the API of interest are recorded.
2) **Detecting concepts.** We use a clustering technique to automatically detect groups of API elements that are often added together (these groups are called *concepts*).
3) **Linking concepts with transactions.** We automatically link concepts to the transactions where they were implemented in the client program. This step is necessary because step 2 only provides concepts expressed as sets of API elements, without temporal information.
4) **Generating patterns.** We generate temporal API usage patterns; each pattern describes a set of concepts that were implemented together in a specific temporal order.

[1] http://hc.apache.org/

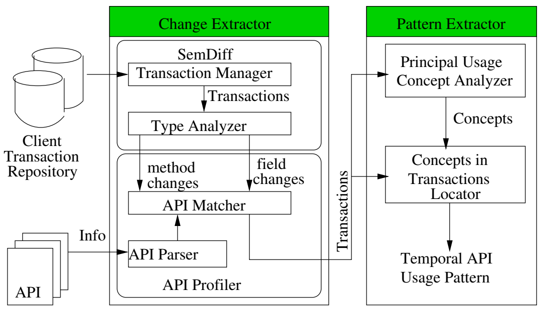

Figure 1: The architecture of the temporal pattern mining system

Figure 1 presents an overview of our infrastructure. We discuss the details of each step in the following sections.

A. Processing Change History

In Figure 1, the left section (`Change Extractor`) shows the tool support for processing change history. We use `SemDiff` [6] to download change differences from client program repositories and aggregate them into *transactions* using standard practices for mining software repositories. In `Type Analyzer`, we resolve lexical tokens that correspond to changed lines of code into fully-resolved symbols using Partial Program Analysis (PPA) [5]. We catalog all the names of the methods and fields of the API of interest (in `API Parser`) and then match those against method and field references added in each transaction. A *delta* is recorded for each reference to an API method or field added in the change history of the client.

The output of this step is a set of transactions $R = \{r_1, ..., r_n\}$. Each transaction $r_i = (\mathrm{ts}_i, \{\delta_{i,1}, ..., \delta_{i,m}\})$ consists of a time stamp (ts) and a set of deltas, where a delta is the name of an API element. In our analysis, we only consider the addition of API elements to a client program, because we are interested to know how a new API functionality (also called *concept*) is introduced/implemented in a client program. For example, let us assume that during the change history of a client, two files are modified as part of a transaction. In the first file, three calls to API method m_a are added, one call to API method m_b is added, and one reference to API field f_c is added. This transaction would then consist of the timestamp along with five deltas: $\{m_a, m_a, m_a, m_b, f_c\}$.

B. Detecting Concepts

In step 2, we identify the API elements whose references were added mostly together. We use Principal Component Analysis (PCA) [11] implemented by the MATLAB library `princomp` to detect the co-added elements. The input to `princomp` is a MxN daily observation matrix O, with M

Table I: Methods used to implement 'multi-protocol connection'

API Element (Methods and Fields)	Coefficient
HttpClient()	0.24394
protocol.Protocol(String, ProtocolSocketFactory, int)	0.24108
protocol.Protocol.registerProtocol(protocol.Protocol)	0.24108
protocol.SSLProcotolSocketFactory()	0.88275

rows corresponding to M days. Each day maps to N API elements in N columns, with each column entry containing the total number of deltas of that API element on that day. For example, suppose an API has two methods m_1 and m_2. During all transactions of day D_1, m_1 was added 3 times and m_2 2 times. During all transactions of day D_2, m_1 was added 0 times and m_2 4 times. We then define: $O = \{D_1(3, 2), D_2(0, 4)\}$. We group transactions by day to have a larger set of API elements to facilitate the detection of concepts that were implemented in more than one transaction. To construct the observation matrix, we pick the first day when at least one element of the API of interest was added in the client program and create a row for that day, such that each column has the number additions recorded for that day. Once the row is constructed, we move to the next available day when at least one element of the API was added in the client program, and repeat the row construction process. The output from PCA is a set of components, with each component representing a set of API elements that were mostly added together.

Table I shows a concept found in client apachejmeter for the HttpClient API (see Section IV). In this concept, the `HttpClient` and `Protocol` classes are initialized and a connection is established. The Coefficient column indicates that when a call to the `Protocol` constructor was added, the `registerProtocol(...)` method call was also added. The call to the protocol connection socket factory constructor, `SSLProcotolSocketFactory()`, has the highest coefficient value, i.e., out of all the possible combinations of the `SSLProcotolSocketFactory()` constructor in different concepts, it appeared 88.2% times with at least one of the 4 methods in Table I. After manual inspection in step 4, we labelled it as *multi-protocol connection*.

C. Linking Concepts with Transactions

In the third step, we link each concept with potentially more than one transaction where the concept was implemented in the client program. To locate the transactions where a concept was implemented, we determine first the days where the concept was detected, and second the specific transactions(s) in the day(s) where the concept was implemented. Algorithm 1 describes the process. The inputs to the algorithm are the concepts and the transactions. The output is a set of concepts, each linked with a set of transactions where the concept was implemented in the client program. Linking transactions to the detected concepts is not a straightforward process, because each component in PCA takes into account the variation in the different number of addition

```
input : $Concepts = \{c_1, c_2, \ldots, c_m\}$,
        $Transaction = \{r_1, r_2, \ldots, r_n\}$
output: Concepts $c$ with implementing transactions $r$,
        $L = \{c_1(r_i \ldots), \ldots, c_m(r_k \ldots)\}$.
1  Total $N$ days in $Transactions$;
2  foreach Concepts $c_j \in Concepts$ do
3      $S = \emptyset$;
4      foreach day $d$ in $N$ days do
5          Compute PCA-Score $v_j$ of day $d$ using
           Equation 1;
6          add the tuple $(v_j, d)$ in $S$;
7          if $v_j > 0$ then
8              foreach Transaction $t \in T$ of day $d$ do
9                  Compute Similarity $s_t$ between $t$ and $c_j$
                   using Equation 2;
10                 if $s_t > \eta$ then add tuple $(c_j, t)$ in $L$;
```
Algorithm 1: Linking concepts with transactions

of calls to API elements. A naive linking based on single occurrences would introduce too much noise (i.e., concepts would become linked to too many transactions, many of which not representing an introduction of the concept). We handle the variations detected by PCA by calculating the *PCA-Score* of each day, and then locating the concepts in transactions by calculating the *similarity* between the concepts and transactions (both discussed below).

We iterate over the list of concepts, compute the *PCA-Score* (using Equation 1) of a day based on each concept and determine whether the score is greater than 0, which indicates that the concept is most likely implemented on that day according to PCA. In Equation 1, v is the PCA-score of a day, b_i is the coefficient associated with API element e_i, Δ_i is the number of deltas of e_i on that day.

$$v = b_1(\Delta_1) + b_2(\Delta_2) + \ldots + b_n(\Delta_n) \tag{1}$$

We then iterate over each transaction of that day and compute the *similarity* between the concept and the element lists added during that transaction. We compute similarity using two metrics: *availability* and *coverage*. *Availability* determines how much of a concept was actually available in a transaction, and *coverage* determines how much of a transaction was actually covered by a concept. If a concept was the only one for a transaction, then the concept will have a value of 1. If two concepts were implemented in a transaction, then availability will be high for both, but coverage will be lower in the first scenario.

$$\text{availability} = \frac{\text{\# of entities in the concept present in the transaction}}{\text{Total \# of entities in the concept}}$$

$$\text{coverage} = \frac{\text{\# of entities in the concept present in the transaction}}{\text{\# of entities in the transaction}}$$

Then, we compute the similarity of a concept against the transaction as follows:

$$\text{similarity} = \alpha * \text{coverage} + \beta * \text{availability} \tag{2}$$

We gave equal weight to both metrics (i.e., $\alpha = \beta = 0.5$) in the linear combination in the absence of any obvious reason to do things differently at this stage of our investigation. If the similarity metric value is above a threshold ($\eta = 0.25$), we assign the transaction to the concept.

We showed the concept *multi-protocol connection* in Table I. When we applied Algorithm 1 in the client apachejmeter, the algorithm pointed at two transactions committed on day '2007-02-03' (at 02:57:00 and 02:58:00). The first transaction added one call to `HttpClient()` and 3 calls to `SSLProcotolSocketFactory()`. The second transaction added one call to the `Protocol` constructor and one to the `registerProtocol(...)` method. Therefore, the score of day '2007-02-03' for this concept was $(0.24394 * 1 + 0.24108 * 1 + 0.24108 * 1 + 0.88275 * 3) = 3.37435$. The availability values for both the transactions are 0.5 (each transaction contained two of the 4 entities found in the concept). The coverage values for both the transactions are 1 (each transaction contained only the entities found in the concept and no other entities from other concepts). Thus the similarity of both the transactions were $(0.5 * 0.5 + 0.5 * 1)$ = 0.75. References to these API elements were added in two other days as well (i.e., they had PCA-score above 0). However, no transactions other than the two on day '2007-02-03' were returned above the threshold for this concept, because not all the API elements of this concept were added together on any other day. Therefore, we automatically assigned the concept to the two transactions of the day.

Our algorithm links concepts with transactions to detect temporal relations between concepts even when multiple concepts are introduced on a given day. To achieve this, we must necessarily link concepts with individual transactions. The trade-off is that we then rely on the assumption that concepts will mostly be introduced within those transactions.

D. Generating Temporal Patterns

After we locate concepts in transactions, we can find the sets of concepts that are implemented mostly together, i.e., in subsequent transactions. Doing so will provide us temporal usage patterns of an API, where a pattern describes a set of concepts that were implemented *in a given temporal order*. A simple example of such a pattern would be "concept A is implemented immediately before concept B." We use Algorithm 2 to generate such patterns. The input to the algorithm is the output from Algorithm 1, i.e., the list of concepts, each linked with a set of transactions in the client program. The outputs of the algorithm are the temporal patterns. The algorithm works as follows.

A *pattern* consists of concepts that have a *small* temporal distance between them. Based on the prior experimentation with different realizations of our approach [19], tolerating long temporal distances combines more concepts into patterns, but results in a combinatorial explosion of patterns demonstrating very little consistency between clients. By

```
input : Concepts with transactions
        $L = \{c_1(r_i \ldots), c_2(r_j \ldots), \ldots, c_m(r_k \ldots)\}$
output: Temporal usage patterns $P = \{p_1, p_2, \ldots, \}$.
$B = \emptyset$;
foreach Concepts $c_j \in L$ do
    foreach Transaction $t$ where $c_j$ was implemented do
        Pattern $p_t = \{c_j\}$;
        foreach Concepts $c_i \in L$ except $c_j$ do
            Find the neighbor concept $c_i$ with temporal distance of 1 or 2;
            Include $c_i$ with transaction timestamp $ts$ in $p_t$;
        add $p_t$ in $B$;
foreach pattern $p \in B$ do
    sort concepts in $p$ in ascending temporal order;
    support = $\frac{\#\text{ of times } p \text{ observed in } P}{\#\text{ of patterns in } P \text{ involving all the concepts of } p}$;
    if $support > \tau$ then Add pattern $p$ in $P$;
```

Algorithm 2: Generating temporal API usage patterns

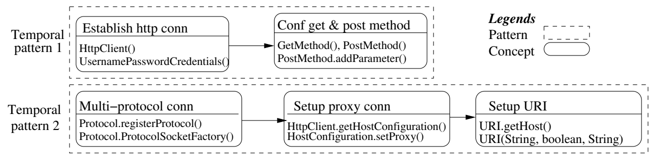
Temporal pattern 1
Establish http conn
HttpClient()
UsernamePasswordCredentials()
Conf get & post method
GetMethod(), PostMethod()
PostMethod.addParameter()
Legends
Pattern
Concept
Temporal pattern 2
Multi−protocol conn
Protocol.registerProtocol()
Protocol.ProtocolSocketFactory()
Setup proxy conn
HttpClient.getHostConfiguration()
HostConfiguration.setProxy()
Setup URI
URI.getHost()
URI(String, boolean, String)

Figure 2: Temporal patterns of HttpClient in client apachejmeter

constraining the introduction time between concepts to a short period, we detect fewer concepts, but more meaningful ones. The temporal distance between two concepts will be 1 if they were implemented in subsequent transactions. For example, if concept A was implemented in a transaction committed at '2010-01-01 09:45:00' and concept B in the next transaction at '2010-01-01 11:20:00', we will say the two concepts have a temporal distance of 1 between them. A concept can be implemented in more than one transaction. For each transaction where a concept is implemented, we find the set of concepts that are within a temporal distance of 1 or 2. We consider each such set as a pattern. Thus, for each concept, we find a set of patterns, from which we pick the final temporal patterns whose support (discussed below) is above a predefined threshold (τ). [2] We calculate support as follows. Suppose we found 10 patterns involving four concepts A, B, C, D. The pattern with order $A \rightarrow B \rightarrow C \rightarrow D$ was found 4 times among those 10 patterns. Then this pattern has a support of 4/10 = 40%.

Figure 2 shows two representative temporal usage patterns observed in client apachejmeter while studying the API HttpClient. Inside each concept, we present two representative methods added in the client to implement the concept. For example, the first concept *establishes an HTTP connection* by calling the `HttpClient` constructor and providing it the

[2]We used the value 0.3 for τ in our experiment. This value has been chosen based on our empirical observation of the data.

necessary credentials. The second concept is the *configuration of GET and POST methods* to read from a web page (using GET) and to submit values to a web page. The second pattern starts with the concept *multi-protocol connection*. The first pattern in Figure 2 showed a support of 100% (i.e., the two concepts in the pattern were always implemented together in the same order in the client program.). The second pattern showed a support of 75%. The three concepts in the second pattern appeared together 4 times, where 3 times they were implemented in the presented order.

E. User Involvement

While the detection and linking of concepts and the generation of patterns are fully automatic, the understanding of a pattern requires human judgement. Using Algorithm 2, we get temporal patterns, such as $\{C1 \rightarrow C10\}$, which puts two concepts $C1$ and $C10$ in temporal order. We must apply manual judgment to determine how, together, the API elements of a concept can be added to achieve a specific functionality. In this process, we find false positives in concepts (i.e., some concepts are not informative as explained in Section V-A). We consider all patterns valid if they have at least two distinct concepts. We experienced patterns, such as $\{C1 \rightarrow \text{false} \rightarrow C1\}$, where $C1$ appeared two times, before and after a spurious concept. Because this pattern contains only one concept, we consider it *uninformative* and discard this pattern. Labeling each concept is a short task when the concepts are cohesive. On average, it took the first author 1-2 minutes to label each concept.

IV. Case Study

This paper reports on our initial attempts at detecting API usage patterns in terms of their introduction in client programs. As such, our initial investigation focused on the feasibility of the idea and potential value of the approach. For this reason, we investigated temporal API usage pattern mining through a *collective case study* [20]. In a collective case study, an issue is selected and analyzed using multiple cases. The issue for this study is *"Whether the temporal usage analysis of an API in different clients will provide useful insights to learn and use the API"*. We selected three cases (*i.e.,* APIs). We adopted an *embedded design* approach by mining temporal patterns from multiple units of analysis (*i.e.,* clients). We studied the following research questions:

- RQ1: What is the precision of the concept detection?
- RQ2: Are patterns consistent between clients?
- RQ3: Can the patterns show dependencies that are not apparent from the structure?
- RQ4: Do patterns represent the decisions of more than one developer?
- RQ5: How can temporal patterns guide improvements in developer documentation?

We selected three APIs that satisfied the *purposeful maximal sampling* process (*i.e.,* they differ enough to illustrate different perspectives on the issue), that were too large to learn within a short time, and that were widely used in different clients. The three APIs, java.util (3030 methods and fields), java.security (1058 methods and fields), and HttpClient (1706 methods and fields) meet these criteria: HttpClient is used to process HTTP,[3] java.security provides functionality to implement secured access to digital resources, and java.util[4] provides collection and utility classes for a wide range of activities.

Table II: Client programs used in the case study

Name	Description	KLOC	Y	API deltas		
				Http	Sec	Util
ant	Build tool	128.3	11	0	60	17638
drools	Rule generator	597.2	5	0	0	2343
groovy	Agile language	110.9	7.5	0	0	1436
spring	App fmk	910	7.5	0	0	20569
jdt	Java IDE	764	8.5	0	0	50550
cdt	C/C++ IDE	963.5	8.5	0	0	53989
eclipsecs	Check style	44.6	7.5	0	0	1269
jadclipse	Decompiler	5.9	9	0	0	248
vuze	Torrent client	465	5.5	0	0	5671
subclipse	SVN client	82.6	6.5	0	0	2373
mylyn	Task manage	128.4	5.5	0	0	27868
htmlunit	Functional test	105.2	10	635	49	9205
rssowl	RSS newsread	100.8	5	34	50	11656
cactus	Unit test	22	11	101	40	2121
jmeter	Load test	2991.8	8	270	23	5309
xsmiles	XML Browser	191	10	340	110	5325
mule	Middleware	1032	6.5	966	220	26617
heritrix	Web crawler	201.7	8.5	484	100	17626
openlaszlo	Web app fmk	91.2	5	30	0	4058

Util =`java.util`, Http = `HttpClient`, Sec = `java.security`
KLOC = Thousand Lines Of non-blank Java Codes, Y = Years active

We selected 19 open source client systems (see Table II) that used at least one of the three APIs, were active for at least 5 years, and that comprised at least 5K non-blank lines of Java code. The clients were developed for a variety of domains, which allowed us to study the properties of the patterns for diverse development needs. We applied our technique on all the clients, e.g., we generated patterns in htmlunit three times, once for each API. On average (unweighted), for HttpClient and per client, our technique produced 30 concepts after step 2 (Section III-B) and 10 patterns after step 4 (Section III-D). After discarding the uninformative patterns (as described in Section III-E), the numbers were 7 patterns per client, each pattern with 2-3 concepts. For java.security, the numbers were 15 concepts, 4 patterns (5 before discarding the uninformative patterns), each pattern with 2-3 concepts. For java.util, the numbers are 58 concepts, 21 patterns (30 before discarding the uninformative patterns), each pattern with more than 5 concepts. However, the java.util concepts and patterns were mostly indistinguishable from each other (discussed in Section V).

[3]We analyzed both the new and old versions, `org.apache.commons.httpclient` and `org.apache.http.client`
[4]Includes all the packages with the prefix `java.util`

Threats to validity. Our evaluation method, the case study, favors in-depth and multifaceted analysis over statistical generalizability (which requires abstracting away the details of the phenomenon). Hence, our assessment will *explain how and why* the approach provides value (or not) in three cases, but will not carry the automatic implication that the same results can be expected in general. Transposing the results to other contexts requires *analytic generalization* [20], namely reasoning about the similarities and contrasts between the cases and other cases of interest. Important factors to consider include whether the API is used in combinations with other APIs, unusual events in the development cycle of the client programs, and the structure of the API.

V. Results

In this section, we present evidence from our three cases to answer our five research questions.

A. What Is the Precision of the Concept Detection? (RQ1)

Mining usage patterns requires tolerating irregularity in the data. Different data mining techniques all offer different trade-offs in how they interpret data to tolerate such irregularity. To study whether PCA offers an appropriate trade-off, we compared our detected concepts against a more general clustering technique, Frequent Itemset Mining (FIM). FIM can be used to group API elements that are introduced mostly together (e.g., in a day). While PCA clusters elements by analyzing the variation in the data (e.g., how two API elements e_1 and e_2 are correlated based on their number of additions into a client in different days), FIM clusters elements based on how frequently the items were introduced together (e.g., whether e_1 was introduced when e_2 was also introduced into the client irrespective of their number of additions). Because our goal is to identify useful patterns, and not to identify all possible valid API usage patterns (for which, there exists no oracle), we focus on precision, and do not take recall into account in our investigation.

Experimental Setup. We ran PCA for each of the client programs as described in Section III-B. Therefore, we ran PCA 35 times (8 client programs for HttpClient, 8 client programs for java.security and 19 client programs for java.util). We ran FIM 35 times on the same data set.[5] For FIM, we pre-processed each of the data sets as follows. For each API, as used in each client program, we created a matrix of dimension $M \times N$ (following the same principle of the algorithm proposed by Agrawal and R. Srikant [2]). Each row corresponds to a day, and each column to an API element. For each element and for each day when the element was added we put a 1 in the column (i.e., element) for that row (i.e., day), otherwise we put a 0. This is different from PCA, in that in PCA we put the total number of times an API element was added in a day. We provide the matrix as input to FIM, and the output is a set of itemsets. Each of the itemsets contains a set of API elements (methods and fields) that were introduced mostly together. Following our definition in Section III, we call these itemsets concepts.

Comparison. For both PCA and FIM, we first removed all concepts of cardinality one from the results, because we do not consider those as concepts (a single API element is unlikely to represent an entire API functionality). Then, for each client, we assessed how many generated concepts represented unique and cohesive concepts and how many were spurious clusters of API elements (false positives). For clients where more than 20 non-singleton concepts were generated, we limited our analysis to the top 20 concepts. If a detected concept is not a false positive, we consider the concept as *informative*. We compute precision as follows.

$$\text{precision} = \frac{|\text{Concepts Detected} \cap \text{Concepts Informative}|}{|\text{Concepts Detected}|}$$

PCA automatically ranks the concepts, where concept 1 accounts for the most variability, i.e., the specific combination of API elements in concept 1 was the most distinct among all concepts, based on their addition into the client program. FIM ranks concepts based on their support, i.e., out of the days, how many introduced a given concept.

We present the precision values for both PCA and FIM in Table III. The manual selection of concepts in our calculation of precision introduces a threat of investigator bias. In practice, we found that spurious concepts are easy to identify. For independent review, we also provide the complete output of both approaches, as well as the result of the manual selection, in an on-line appendix.[6] We computed precision values for each client program for each API, and then took the (unweighted) average of all precision values. For both PCA and FIM, the average precision was the highest for java.security and the lowest (by far) for java.util. In java.util, the precision value was excessively low because almost all the concepts were bloated with collection methods and the usage of other methods was hardly visible. The results clearly show that PCA outperforms FIM for generating concepts. The difference is considerable for HttpClient and java.security, but not for java.util. For both HttpClient and java.security, more than 80% of the top concepts produced by PCA contained 3 to 5 API elements, whereas more than 80% of the FIM top concepts were of length 1 to 3. The precision value of FIM was lower than PCA, because the FIM concepts were mostly grouping API elements as they appeared in the day, instead of accounting for their variance in the data (e.g., if a set of API elements was added in only one transaction, it will have lower support based on FIM, whereas the set may be added to implement an important concept). For example, we present the concept *setup proxy* in Figure 3. We show three versions of the concept, where (A) contains all the appropriate API elements, (B) and (C)

[5]We used the FP-Growth library from rapidminer (http://rapid-i.com/)

[6]http://www.cs.mcgill.ca/~swevo/icse2012g.zip

Table III: Precision values for PCA and FIM

	PCA			FIM		
API	**Avg**	**StdDev**	**Max**	**Avg**	**StdDev**	**Max**
HttpClient	0.77	0.08	0.85	0.49	0.09	0.60
Security	0.78	0.14	1	0.63	0.13	0.80
Util	0.26	0.08	0.40	0.20	0.08	0.35

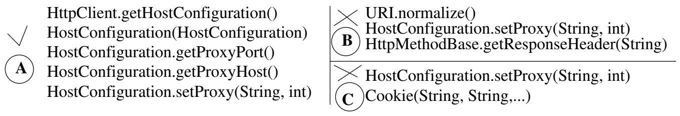
HttpClient.getHostConfiguration()
HostConfiguration(HostConfiguration)
HostConfiguration.getProxyPort()
A
HostConfiguration.getProxyHost()
HostConfiguration.setProxy(String, int)
URI.normalize()
HostConfiguration.setProxy(String, int)
B
HttpMethodBase.getResponseHeader(String)
HostConfiguration.setProxy(String, int)
C
Cookie(String, String,...)

Figure 3: The API elements added to construct the concept *setup proxy*. (A) is the ideal candidate. (B) and (C) are false positives. (A) and (B) were detected by PCA and (C) was detected by FIM.

do not contain all the API elements. (A) and (B) were detected by PCA and (C) was detected by FIM. We found that the API elements of (A) were added together in a transaction, whereas the API elements of (B) and (C) were added together more than once in different transactions. Even though PCA incorrectly detected the concept (B), it also detected that API elements of (A) were added strictly together in one transaction, and thus that may represent an interesting data variation point. Because FIM relies on the support of a group, it ignored the group (A).

B. Are Patterns Consistent Between Clients? (RQ2)

We consider a pattern as consistent if the same set of concepts were implemented in the same temporal order in more than one client program. We calculate consistency as follows. For each such pattern of an API,

$$\text{consistency} = \frac{\text{\# of clients exhibiting the pattern}}{\text{Total \# of clients that used the API}}$$

For example, if the establishment of an HTTP connection always precedes an HTTP status error checking in all client programs, then it has a consistency of 100%. In Table IV, we summarize our findings about the consistency of the patterns (after discarding uninformative patterns as discussed in Section III-E). For each API, we show how many patterns were found consistent and how many were inconsistent. We consider a pattern as consistent, if the pattern was visible in at least two client programs. We provide statistics on the consistency for the three APIs in the client programs by presenting the unweighted average of the consistencies of all the patterns, the standard deviation of the consistency values, their minimum and maximum values. The consistency values are generated by taking into account the consistency values of only the consistent patterns.

The maximum consistency value of all the patterns in the two APIs HttpClient and java.security is 1, i.e., at least one consistent pattern in each API was found in all the client programs where the API was used. The number of consistent patterns in java.util was the lowest (only 1). For instance, java.util concepts mostly contained the collection framework

Table IV: Pattern consistency across client programs

	Patterns		Consistency			
API	**#Cons**	**#Incons**	**Avg**	**StdDev**	**Min**	**Max**
HttpClient	12	5	0.56	0.29	0.25	1
Security	5	4	0.68	0.26	0.38	1
Util	1	>100	0.16	0	0.16	0.16

methods (`ArrayList` and `HashMap` methods) in no specific order. Thus, most of the patterns in java.util described different combinations of concepts, where the concepts only involved the collection framework methods. These methods are so general-purpose that it is impossible to find the rationale behind using those concepts (e.g., why was this array created and this item was added into the array?). The one consistent pattern was related to the implementation of regular expression functionality based on the `regex` library methods, which was mostly preceded and followed by the implementation of array or hash data structures (in 3 out of 19 clients).

In Figure 4, we show the 8 most consistent patterns for HttpClient, where each pattern was found at least in 50% of its client programs. Each pattern is separated by a horizontal line. The patterns are placed in the relative order based on how they appeared in the client programs. For example, pattern 1 appeared every time the API was first introduced in a client program. Pattern 2 appeared after pattern 1 in 5 out of 7 client programs, etc. The rectangle to the left of each pattern shows the consistency of a pattern in the client programs. For example, pattern 1 was found in 8 out of the 8 client programs that used the HttpClient API, pattern 2 was found in 7 out of 8 client programs. We show 2 representative API elements for each concept, although a concept can have more than 2 elements. The client programs, mule, heritrix, xsmiles and jmeter implemented most of these patterns which, we note, are from three different domains (enterprise middleware, web crawler, and testing frameworks). We show similar patterns for java.security in Figure 5, where again the client programs xsmiles, mule, heritrix and jmeter implemented all these patterns.

C. Can the Patterns Show Dependencies That Are Not Apparent From the Structure? (RQ3)

When manually inspecting an API, a developer can sometimes figure out the order in which certain elements must be added. For example, if the developer wants to call method B.m1(A a), it is clear that an instance of type A must be obtained before calling B. As partial evidence that the patterns we detected could be useful, we investigated whether they included relations between elements that were not directly apparent from the structure. For example, in Figure 4, 50% of the clients exhibit pattern 8, which involves three concepts, setup cookie, setup session and setup proxy. According to the pattern, the setup of a HTTP-based session should follow an initial setup or check of cookie. This

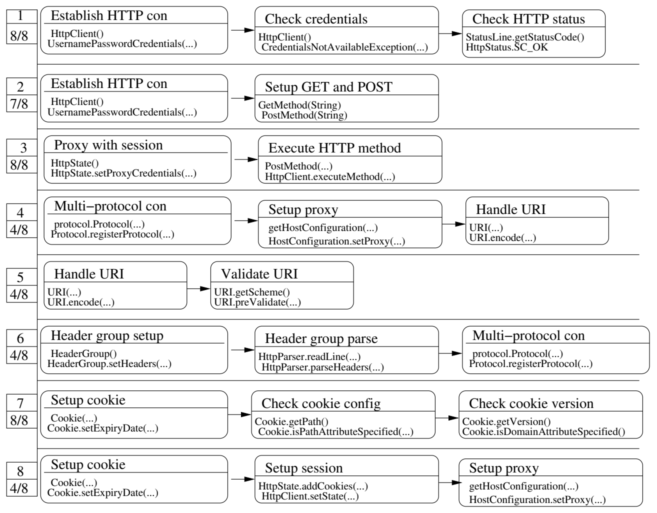

Figure 4**:** Most consistent patterns for the HttpClient API.

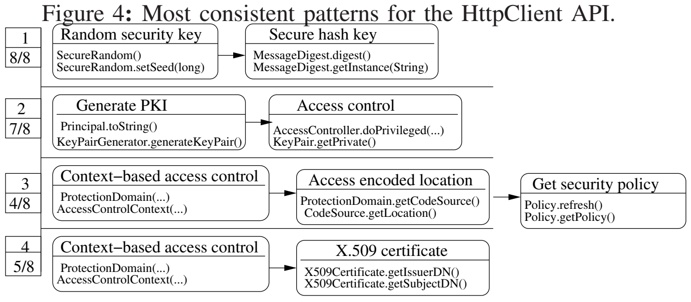
1
8/8
Random security key
SecureRandom()
SecureRandom.setSeed(long)
Secure hash key
MessageDigest.digest()
MessageDigest.getInstance(String)
2
7/8
Generate PKI
Principal.toString()
KeyPairGenerator.generateKeyPair()
Access control
AccessController.doPrivileged(...)
KeyPair.getPrivate()
3
4/8
Context−based access control
ProtectionDomain(...)
AccessControlContext(...)
Access encoded location
ProtectionDomain.getCodeSource()
CodeSource.getLocation()
Get security policy
Policy.refresh()
Policy.getPolicy()
4
5/8
Context−based access control
ProtectionDomain(...)
AccessControlContext(...)
X.509 certificate
X509Certificate.getIssuerDN()
X509Certificate.getSubjectDN()

Figure 5**:** Most consistent patterns for the API, java.security

dependency is not evident from the structure of the two corresponding classes, `Cookie(...)` and `HttpState(...)`. However, the pattern shows that the setup of session also adds cookies (`HttpState.addCookies(...)`). This pattern shows that temporal patterns can describe non-obvious dependencies.

Formally, we determine whether a pattern exhibits hidden order as follows. First, we determine the number of dependencies in a pattern. For example, if pattern P is $C1 \rightarrow C2 \rightarrow C3$, then P shows two dependencies between the three concepts. We consider that a method $m2$ in $C2$ is related to a method $m1$ in $C1$ if the set $\{$target type, param types$\}$ of $m2$ intersects with the return types of $m1$. If the methods of $C1$ and $C2$ are related, we consider the dependency between them as structural, otherwise the dependency is hidden. We investigated two options: we consider two concepts as structurally dependent only when 1) at least 50% methods are related between them, or 2) at least one method was related between them. We implemented this procedure in a script[7] and ran the script on the consistent patterns of HttpClient (Figure 4) and java.security (Figure 5) using options: threshold = 0.5 and threshold = 0. We did not investigate java.util further, because it had only one consistent pattern, so its result will not provide us useful insight. In Table V, we present the results. For HttpClient,

[7]Available in our online data (See footnote 6)

Table V**:** Surprisingness values of the APIs

API	Threshold	Structural	Hidden	Surprisingness
HttpClient	0.5	1	12	0.92
	0	6	7	0.54
Security	0.5	0	5	1
	0	3	2	0.4

Table VI**:** Diversity values of the APIs

Pattern	M	D	S	Pattern	M	D	S
H1 (Conn)	2	1	7	H7 (Cookie)	3	2	6
H2 (Method)	2	1	6	H8 (Session)	3	2	2
H3 (Proxy)	4	1	7	S1 (Secure key)	2	2	6
H4 (Protocol)	3	2	2	S2 (PKI)	5	3	5
H5 (URI)	3	2	2	S3 (Access)	3	2	2
H6 (Header)	3	3	1	S4 (X.509)	3	2	2

P = Pattern ID (H for HttpClient, S for java.security)
M = Maximum # developers implemented the pattern in a client
D = # clients implemented the pattern using multiple developers.
S = # clients implemented the pattern using one developer.

there were 13 dependencies in the 8 consistent patterns, 12 of which were found as *hidden* for the threshold = 0.5. The column 'Surprisingness' was calculated as follows.

$$\text{surprisingness} = \frac{\text{\# Hidden Dependencies}}{\text{\# Total Dependencies}}$$

The surprisingness value was the highest for java.security (5 out of the 5 dependencies were hidden with threshold = 0.5). When applied with no threshold, 3 out 5 dependencies of java.security and 6 out of 13 of HttpClient dependencies were found structural. Nevertheless, the results show that non-obvious relations do exist between API functionality and temporal patterns can uncover such useful information.

D. Do Patterns Represent the Decisions of More Than One Developer? (RQ4)

The patterns we detect would not be of general interest if they only represent a single developer's idiosyncratic use of an API. To ensure that this is not the case, we also investigate whether our patterns were introduced by more than one developer. If a pattern was introduced by more that one developer, we consider it further evidence of the generalizability of the pattern. We find the number of developers implementing a pattern by first determining how many developers have implemented each concept of the pattern and then summing the total number of distinct developers thus associated with the pattern.

In Table VI, we present the results for all consistent patterns of HttpClient (Figure 4) and java.security (Figure 5). Each of the patterns is denoted by an ID and a short description. For example, the first pattern of Figure 4 is denoted as 'H1' with a description of its primary functionality (i.e., establishing HTTP connection). The column 'M' shows the maximum number of developers found in a single client program to implement the pattern, 'D' shows how many client programs used more than one developer to implement the pattern and 'S' shows how many client programs used only one developer. The results show that all the patterns were implemented by more than one developer in at least one client program. For example, for HttpClient, H3 was implemented by the most number of developers (4) in a client program (mule), whereas for java.security, S2 has the maximum number of developers (5).

E. How Can Temporal Patterns Guide Improvements in Developer Documentation? (RQ5)

We compared the documentation of HttpClient[8] and java.security[9] with the temporal patterns of Figure 4 and Figure 5, respectively. Our focus was to investigate whether the documentation correctly represents the API usage found through our temporal patterns and whether and how it has the potential to improve the learnability of the documentation. We located in which section of the documentation each pattern was described. Then, we compared the order in which the concepts were presented in the documentation with the order in which they were implemented, as found by our temporal patterns. Ideally, the documentation should refer to all concepts in the same section in the same order as the temporal pattern. If at least 50% of the API elements of a concept were found in a section, we determine that the concept was discussed in the section. If the concept is part of a pattern, we say that we can map the pattern to the section. If more than one concept is found in the section, we investigate whether the concepts are mentioned in the same order as in the pattern. Finally, if the concepts of a pattern are discussed in many sections, we say that the documentation of the pattern is *scattered*.

In Figure 6, we show a high-level overview of the comparison between the HttpClient patterns generated as shown in Figure 4 and the HttpClient documentation. For the documentation, we only show the sections that present different API functionality with clear rationale in the title(s) (e.g., section 1 is shown because it specifically discusses the establishment of an HTTP connection with credentials). We observe that 50% of the patterns (1, 3, 7, 8) presented in Figure 4 were matched in the documentation. The patterns found could be used to improve the organization of the documentation. For example, pattern 2 was briefly discussed using examples in section 11. The pattern 1 was scattered in three sections (1, 4, 11). Because this pattern was always implemented when the HttpClient API was introduced in a client, it may be best to present this pattern in section 1.

The java.security documentation maps to two patterns (1, 4) of Figure 5 and provides a theoretical overview of the other two patterns, without referring to specific API types. The concept 1 of pattern 2 was discussed as a separate

[8]http://hc.apache.org/httpclient-legacy/index.html

[9]http://download.oracle.com/javase/6/docs/technotes/guides/security/overview/jsoverview.html

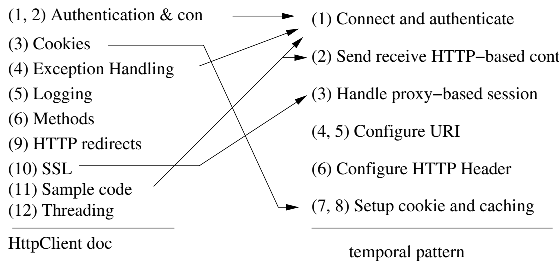
(1, 2) Authentication & con
(3) Cookies
(4) Exception Handling
(5) Logging
(6) Methods
(9) HTTP redirects
(10) SSL
(11) Sample code
(12) Threading
HttpClient doc
(1) Connect and authenticate
(2) Send receive HTTP−based contents
(3) Handle proxy−based session
(4, 5) Configure URI
(6) Configure HTTP Header
(7, 8) Setup cookie and caching
temporal pattern

Figure 6: Comparison of HttpClient temporal patterns (ref: Figure 4) and their corresponding sections in the documentation.

section (5). The access control mechanism was part of 3 patterns in Figure 5, but it was covered in one single section (6).

Temporal patterns can help identify deficiencies in the documentation such as sections that should mention concepts that are added together or temporal patterns that are frequently used but that are not documented at all. Overall, comparing the documentation with the temporal usage patterns took us less than one hour. This review process is more efficient than collecting subjective user opinions on mailing lists over years, as it is the case in open source projects [7].

VI. Discussion

Detecting API usage patterns in terms of their introduction into client programs is an ambitious goal. In particular, given the inexact nature of the patterns detected, the development of the technique is open to an endless number of potential design decisions and optimizations. Our initial exploration into the matter demonstrated the basic feasibility of the idea. In addition, it has taught us a number of useful lessons for the further development of the technique.

Sliding window. Because the grouping of transactions by day is only a temporary step to improve the PCA-performance, we opted for simplicity and used an absolute classification based on the system clock (as opposed to a sliding window [23]). For our preliminary investigation, we did not deem necessary the additional conceptual complexity of a sliding window, since in step 3 we trace concepts back to individual transactions. In practice, we found less than 1% of the API-transactions fell in the midnight-region (± 3 hours), so midnight was a natural boundary for the vast majority of development days in our cases. Moreover, in our particular cases, there was often a considerable gap between the days when transactions were committed. For systems with a much denser stream of transactions, it may be worth investigating the use of a sliding window for initial clustering.

Concept detection window and thresholds. While we grouped transactions in days to form a better detection of concepts, we could have used different window sizes. We experimentally applied our technique with 5 and 7-day windows. We found that the concepts are (predictably) more scattered and indistinguishable for these window sizes. This was because when bundled in 5 or 7 day windows, many concepts were indistinguishable from each other based on their data variation. We also tried to detect concepts in individual transactions, but that returned too many concepts which were just clones of each other. In the end, a 1-day window produced the best results. Again, for systems with radically different transaction streams, preliminary investigation may determine that a different window size is better suited. The other two thresholds (η and τ) were also established experimentally, and as part of our future work we plan to systematically study their impact on the results.

The nature of the API. Even though we studied the API java.util in 19 client programs, we were unable to find particularly useful patterns for it. This API provides collection framework methods that were added for general usage. The concepts of this API were so overly bloated with those methods that it was impossible to determine the implementation intent of a pattern. We believe our approach is more suitable for APIs that are focused towards specific functionality (e.g., security enforcement or HTTP connections). A possible way to extend our approach to handle APIs like java.util would be to remove collection framework methods as stop words (following the IR principle [14]).

VII. Related Work

The automated inference of API usage properties has benefited from considerable attention in recent years. In the space available, we must inevitably limit our discussion to a selection of representative works.

Non-temporal API usage pattern mining. Extensive work has targeted the inference of patterns from client programs. Many such techniques consider each function/method as a data instance with attributes such as methods called, fields accessed, types used, etc. Machine learning algorithms can be applied to such data to infer patterns. As representative examples, we note the use of association rule mining by Michail [15], research on code example recommenders [4, 10], and programming rules inference tools such as PR-Miner [12]. In these approaches, the evolution of clients is not considered and their temporal relations are not analyzed.

Client evolution analysis. A different class of techniques analyzes the *evolution* of APIs in target systems. For example, Zimmermann el al. [24] and Ying et al. [21] developed techniques to recommend classes (or methods) that should be changed together based on the inference of association rules defined over the transactions stored in the client's revision control system. Alam et al. [3] analyzed the insertion, deletion, and modification of functions in client systems to discover sets of changes that temporally affect future development. In this class of work, the unit of analysis is the *transaction*, as opposed to individual functions/methods. This is also the model we follow to detect temporal API usage patterns. However, the research cited above focuses on changes in client software, but without consideration for any API element actually used in the change. In contrast, we further process the change data to automatically discover the number and type of references to API elements, and perform our analysis on the API usage data.

Temporality in API usage analysis. Pattern inference techniques have also been proposed to analyze temporal aspects in API usage. As an example of a technique based on *static analysis*, Acharya et al. [1] use model-checking technology to automatically generate static traces representative of potential program paths through client functions, then extract scenarios (subsets) from traces, mine the resulting traces for partial orders, and derive final specifications by removing spurious patterns through a consistency check across multiple clients. Temporal usage patterns can also be detected through *dynamic analysis*. For example, Lo et al. [13] generate the traces to be mined for temporal patterns through the instrumentation and subsequent execution of client programs. In such rule-mining approaches, the notion of temporality relates to *order within a client function*, e.g., the detection that a `lock()` method should precede `unlock()` on all program paths. In contrast, our notion of temporality relates to order of *introduction in the development history*. A further difference is that we analyze relations between groups of API elements (concepts), as opposed to between individual elements.

Finally, Mileva et al. determine the popularity of an API based on its overall usage in different clients [16]. They also looked at the different versions of an API to find whether a version is more popular than other versions [17]. They study trends in the use of an entire API, whereas our work analyses usage at the element level.

VIII. Summary

We developed a technique that analyzes the evolution of API usage in client programs. We identify temporal usage patterns, *i.e.,* sequences of concepts in the client programs. We evaluated our technique by analyzing three APIs in up to 19 open source client programs. We found that client programs of java.security and HttpClient mostly shared a consistent set of usage patterns, even if the clients did not provide similar features. As we discussed in Section V, these patterns can uncover interesting hidden dependencies between API functionality (not apparent from their structural dependencies) and can improve API documentation. We also observed that an API that did not require complex interactions between its elements such as java.util, did not induce common temporal usage patterns in client programs.

Acknowledgments

The authors are grateful to Doina Precup for her advice on PCA, and to Luke Bayly for his contributions to the data collection tools. This work is funded by NSERC and IBM.

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *Proc. 6th Joint Meeting of the European Software Engineering Conf. and ACM SIGSOFT Symp. on the Foundations of Software Eng.*, pages 25–34, 2010.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. Conf. of the 20th Int. Conf. on Very Large Databases*, pages 192–202, 1994.

[3] O. Alam, B. Adams, and A. E. Hassan. A study of the time dependence of code changes. In *Proc. 16th Working Conf. Reverse Eng.*, pages 21–30, 2009.

[4] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proc. 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222, 2009.

[5] B. Dagenais and L. Hendren. Enabling static analysis for Java programs. In *Proc. 23rd Conf. on Object-Oriented Prog. Systems, Languages and Applications*, pages 313–328, 2008.

[6] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proc. 30th Intl. Conf. Soft. Eng.*, pages 481–490, 2008.

[7] B. Dagenais and M. P. Robillard. Creating and evolving developer documentation: Understanding the decisions of open source contributors. In *Proc. 18th Intl. Symp. Foundations of Soft. Eng.*, pages 127–136, 2010.

[8] U. Dekel and J. D. Herbsleb. Improving API documentation usability with knowledge pushing. In *Proc. 31st Intl. Conf. Soft. Eng.*, pages 320–330, 2009.

[9] E. Duala-Ekoko and M. P. Robillard. Using structure-based recommendations to facilitate discoverability in APIs. In *Proc. 25th European Conf. Object-Oriented Prog.*, pages 79–104, 2011.

[10] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Soft. Eng.*, 32(12):952–970, 2006.

[11] I. Jolliffe. *Principal Component Analysis*. Springer Series in Statistics, 2nd edition, 2002.

[12] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. 10th European Soft. Eng. Conf. and Intl. Symp. Foundations Soft. Eng.*, pages 306–315, 2005.

[13] D. Lo, S.-C. Khoo, and C. Liu. Mining temporal rules for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):227–247, 2008.

[14] C. D. Manning, P. Raghavan, and H. Schütze. *An Introduction to Information Retrieval*. Cambridge University Press, 1st edition, April 2009.

[15] A. Michail. Data mining library reuse patterns in user-selected applications. In *Proc. 14th Intl. Conf. Automated Soft. Eng.*, pages 24–33, 1999.

[16] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller. Mining trends of library usage. In *Proc. Intl. ERCIM Workshop on Principles of Soft. Evolution and Software Evolution Workshop*, pages 57–62, 2009.

[17] Y. M. Mileva, V. Dallmeier, and A. Zeller. Mining API popularity. In *Proc. Conf. of Testing: Academic & Industrial Conf. Practice and Research Techniques*, pages 173–180, 2010.

[18] M. P. Robillard and R. DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 2011. DOI: 10.1007/s10664-010-9150-8.

[19] G. Uddin, B. Dagenais, and M. P. Robillard. Analyzing temporal API usage patterns. In *Proc. 26th IEEE/ACM Intl. Conf. on Automated Software Engineering*, page 4, http://www.cs.mcgill.ca/~martin/papers/ase2011.pdf, 2011. To appear.

[20] R. K. Yin. *Case study Research: Design and Methods*. Sage, 4th edition, 2009.

[21] A. T. Ying, G. C. Murphy, R. Ng, , and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Soft. Eng.*, 30(9):574–586, 2004.

[22] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proc. 23rd European Conf. Object-Oriented Prog.*, pages 318–343, 2009.

[23] T. Zimmermann and P. Weiβgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. 1st Intl. Workshop on Mining Software Repositories*, pages 2–6, 2004.

[24] T. Zimmermann, P. Weiβgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.