Analyzing Temporal API Usage Patterns

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

***Abstract*—Software reuse through Application Programming Interfaces (APIs) is an integral part of software development. As developers write client programs, their understanding and usage of APIs change over time. Can we learn from long-term changes in how developers work with APIs in the lifetime of a client program? We propose *Temporal API Usage Mining* to detect significant changes in API usage. We describe a framework to extract detailed models representing addition and removal of calls to API methods over the change history of a client program. We apply machine learning technique to these models to semi-automatically infer temporal API usage patterns, i.e., coherent addition of API calls at different phases in the life-cycle of the client program.**

***Index Terms*—API Usage, API Usability, Usage Pattern, Software Reuse, Mining Software Repositories.**

I. Introduction

Software systems often reuse functionality provided by libraries and frameworks. The software, as a client program, reuses functionality through Application Programming Interface (API).

Despite advances in API documentation and assisting technologies [1, 4, 14], large APIs are still hard to learn [11]. A major challenge for API users is to discover the subset of the API that can help complete a task [11]. 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 V), existing technology does not provide guidance about *when* certain patterns are relevant in the life-cycle of client program, or whether there exist *temporal relations* between different patterns.

In this paper, we propose *Temporal API Usage Pattern Mining*. Our notion of temporal API usage is founded on the observation that *cohesive subsets of an API are often integrated into a client program at different points in the lifetime of the client program*. We based our pattern detection strategy on the mining of the change history of API client programs and the analysis of the addition and removal of calls to API methods as part of transactions in a revision control system. We developed a framework that can produce a description of the groups of API elements (typically, methods), that are integrated into client programs at roughly the same time in the history of the client program, along with the *order* between these groups of elements. First, we automatically partition the life-cycle of a client program into a set of usage phases based on the usage of an API in the client program. We then employ a machine learning technique on the API usage data to identify principal usage components, where a usage component comprises a set of related API elements that were used mostly together to implement a particular functionality. We finally map the usage components to the detected phases to generate the temporal usage patterns.

Knowledge of temporal API usage patterns can provide many benefits to software development organizations. For instance, this knowledge can inform developers on the next possible steps in using an API. Temporal usage patterns would thus add a new dimension to existing API recommendation systems such as API Explorer [5] and Intelligent Code Completion Systems [2]. Temporal API usage patterns can also be used to improve developer learning resources, for example by ensuring that tutorials cover each scenario in order.

This work is our first step in the exploration of the potential of temporal API usage patterns.

II. Temporal API Usage

We observed the following temporal usage pattern scenario during our preliminary investigation of client programs using the HttpClient library. While initializing a HTTP connection using the HttpClient API,[1] a developer might first add a call to the `HttpClient` constructor, and also add a call to the methods of the class `UserNamePasswordCredentials`, with user name and password. 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` and `HttpStatus.SC_NOT_FOUND`. In later development, the developer may call the methods of `GetMethod` to retrieve contents from the server. We are interested in detecting the various phases of development, the API elements used in each phase, and the ordering between phases. In our example, one client program may establish the connection in the first phase, and then implement error handling in the next phase, whereas another client program may establish the connection first and then message retrieval and posting configurations in the next phase. However, in both cases, a connection to the HTTP server is required to proceed to the next phase.

For a given API and client program, the structure of temporal usage patterns we detect corresponds to a *sequence*

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

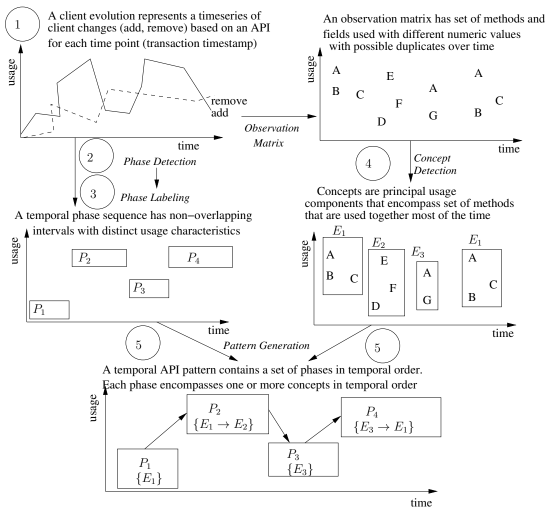
1
A client evolution represents a timeseries of client changes (add, remove) based on an API for each time point (transaction timestamp)
usage
remove
add
time
Observation Matrix
An observation matrix has set of methods and fields used with different numeric values with possible duplicates over time
usage
A B C E F D A G A B C
time
2
Phase Detection
3
Phase Labeling
A temporal phase sequence has non−overlapping intervals with distinct usage characteristics
usage
P_1 P_2 P_3 P_4
time
4
Concept Detection
Concepts are principal usage components that encompass set of methods that are used together most of the time
usage
E_1 A B C
E_2 E F D
E_3 A G
E_1 A C B
time
5
Pattern Generation
5
A temporal API pattern contains a set of phases in temporal order. Each phase encompasses one or more concepts in temporal order
usage
P_1 $\{E_1\}$
P_2 $\{E_1 \rightarrow E_2\}$
P_3 $\{E_3\}$
P_4 $\{E_3 \rightarrow E_1\}$
time

Fig. 1. Temporal API Usage Analysis Framework

of phases $\{P_1, \ldots, P_n\}$ where a phase object $P_i = (l, C)$ is a tuple comprising a *phase label* l and a *set* $C = \{c_1, \ldots, c_m\}$ of groups c_i of API elements (methods and fields) that tend to be used together in a phase (but not necessarily in the same revision control transaction). We call such groups of elements *concepts*. For example, for an API A used in a client program S, our approach can detect the temporal usage pattern:

$$\begin{aligned} T_{A,S} = \quad & \{(\text{Intro}, \{\{m_1, m_2\} \rightarrow \{m_3, m_4\}\}), \\ & (\text{Cleanup}, \{\{m_3, m_4\}\})\} \end{aligned}$$

which indicates that an *Intro* phase was detected where groups of methods $\{m_1, m_2\}$ and $\{m_3, m_4\}$ were introduced and the group $\{m_1, m_2\}$ was introduced before the group $\{m_3, m_4\}$. The intro phase was followed by a *Cleanup* phase, where a group of methods was involved (typically, removed).

III. Analysis Framework

The input to our approach is the source history of a client program that uses the API of interest and the output is a temporal usage pattern as described above.

Figure 1 shows the steps that we perform to mine the temporal API usage patterns. The arrows from the top to bottom show the flow of the analysis. We incorporate the execution step number of each arrow inside a circle. First, we collect client program history with respect to the API of interest, where each change in the client program is either an addition of the API element inside the client program or a removal of an API element from the client program at a particular time. Second, we automatically partition the timeseries of client program changes into a number of distinct phases. Each phase contains an interval with a start day and an end day indicating distinct usage characteristics in the client program based on the API. For example, a phase may indicate a significant increase in the usage of the API over a short interval, when several API functionality were introduced in the client program. Third, we label the phases based on a number of heuristics, e.g., a phase with a high churn would be labelled burst. Fourth, we detect co-used API elements in the client program change data and identify the underlying concept that was implemented using the API elements. Fifth, we automatically locate the concepts that were implemented inside each usage phase and generate the temporal usage pattern, where each phase contains one or more concepts in temporal order.

Each of the five steps are supported by our infrastructure. In the following subsections, we briefly explain the techniques behind the five steps. All steps are fully automated except the fourth step, where human guidance is necessary.

Step 1: Change history processing. We convert the unstructured textual differences between files versions in the client program using SemDiff [3] into a database of transactions where all additions and removal of references to methods and fields of the API of interest are recorded.

The output of this step is a set of transactions $R = \{r_1, ..., r_n\}$. Each transaction $r_i = (\text{ts}_i, \{\delta_{i,1}, ..., \delta_{i,m}\})$ consists of a time stamp (ts) and a set of deltas, where a delta $\delta = (a, e)$ is an action $a \in \{+, -\}$ and e, the name of an API element. The action indicates whether a reference to the API element e was added or removed as part of the delta. For example, let us assume that during the change history of a client program, 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 removed, and one call to API field f_c is added. This transaction would then consist of the timestamp along with five deltas: $\{(+, m_a), (+, m_a), (+, m_a), (-, m_b), (+, f_c)\}$.

Step 2: Phase detection. We split the change history of the client program into different *phases* (non-overlapping intervals). Intuitively, a phase represents a period of development time that does not present any major discontinuities in the amount of interactions with the API of interest that is added or removed from the client program.

We partition the changes related to the API to identify usage phases. We detect each phase interval by analyzing the deltas in a client program using a sliding window-based approach. The inputs to the algorithm are the *daily totals* for each day during which the API was used in transactions in the client program. The output is a set of non-overlapping phase intervals, each containing a start day and an end day.

The daily total of a day is the total number of deltas for that day, which includes all the transactions whose time-stamp falls within the 24-hour period corresponding to that day. The window size is 90 days, and the sliding window step size is 7 days. We determined these thresholds based on early experimentation with the approach.

We start with the first day when the API was first introduced in the client program, perform the sliding window-based interval detection approach, and then move to the day that was 7 days later. At each step, we compute the mean and standard deviation of the daily totals for the window starting from the day to the next 90 days. If there is a significant deviation (e.g., the mean+standard deviation of the present window is twice bigger than the mean+standard deviation of the previous window) in terms of API usage detected at a step, we identify that as a potential phase interval based on our algorithm.

Step 3: Phase labeling. We automatically (and heuristically) label each phase in terms of the type of interactions with the API that best represent the phase. We determined five categories based on our experience and empirical observation: *Intro*, *BuildUp*, *Burst*, *Incremental* and *Cleanup*. A phase can appear multiple times in the lifetime of a client program, and we do not expect the sequence of phases to be the same for all client programs. Intuitively, we define the phases as follows. *Intro* starts when the API is first introduced. During the *Burst* phase, new API references are added and the API is accessed more intensely than in other phases. The *BuildUp* phase shows an upward increase in the addition of new API references, but not as high as the *Burst* phase. During the *Incremental* phase, the amount of new API references introduced may be below the above mentioned three phases. During the *Cleanup* phase, API interaction consists mostly of removal of API references.

Our phase labeling heuristics take into account three metrics: New functionality Added (α), Total Usage (β), Add to Remove (γ).

- **New Functionality Added (α):** This metric captures the importance of a phase in terms of new API references to the client program. The Burst and BuildUp phases are expected to have larger α values. In Equation 1, n_i is the total number of new API references during phase i, and K is the total number of phases detected over the client program's lifetime.

$$\alpha = \frac{n_i}{\sum_i^K n_i} \tag{1}$$

- **Total Usage (β):** We measure the fraction of deltas (i.e., addition or removal of API references) observed in a phase with respect to the total deltas committed over the client program's lifetime. We expect this metric to be higher for a Burst phase. An Incremental phase may show a high value as well, if it is very long (e.g., at least 6 months). In Equation 2, p_i is the total deltas committed inside a phase i, and K is the number of phases detected over the client program's lifetime.

$$\beta = \frac{p_i}{\sum_{i=1}^K p_i} \tag{2}$$

- **Add To Remove (γ):** We measure the contribution of a phase in terms of the retained deltas. We expect this metric to have a value close to 0 for a Cleanup phase, with minimal adds but higher number of removals in deltas. In Equation 3, a is the total number of deltas added and r is the total number of deltas removed inside a phase.

$$\gamma = \frac{a}{r} \tag{3}$$

Step 4: Concept detection. We use a machine learning technique on the transactions retrieved in step 1 to automatically detect groups of API elements that are often used together (these groups are called *concepts*). We then manually label the concepts. We use the machine learning approach Principal Component Analysis (PCA) [7] to detect the co-used elements, e.g., in particular, more than one element of HttpClient might be used to implement a concept.

We employed the MATLAB tool princomp[2] to perform PCA. The input to princomp is a MxN daily observation matrix O, with M 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 $m1$ and $m2$ and was used in days $D1$ and $D2$. During day $D1$, $m1$ was called (added and removed) 3 times and $m2$ 2 times. During day $D2$, $m1$ was called 0 times and $m2$ 4 times. We then define: $O = \{D1(3,2), D2(0,4)\}$. We group transactions by day to have a larger set of API elements than a transaction and to facilitate the detection of concepts that were implemented in more than one transactions. The output from PCA is a set of components, with each component representing a set of API elements that were mostly used together. We term those components as concepts by manually tagging the components based on the coherence of the API functionality between the elements of each component (e.g., establishing a HTTP connection).

Step 5: Pattern generation. We automatically assign concepts to phases and generate the final temporal API usage pattern. We generate temporal usage pattern of an API in a client program by placing each detected concept inside the phase(s), where the concept was implemented. The inputs to the algorithm are the phases and the concepts. The output is a temporal usage pattern as described in Section II. The algorithm runs for every detected phase and determines which concepts were implemented inside the phase. We determine the implementation day of a concept inside a phase by automatically locating the day when the concept was implemented inside the phase.

IV. Case Study

We are currently investigating the effectiveness of our approach and the usefulness of temporal usage patterns using a *collective case study* [12]. In a collective case study, an issue is selected, and it is analyzed using multiple cases. The issue for this study is *"Whether the temporal usage analysis of an API in different client programs will provide useful insights to learn and use the API"*. We have selected multiple cases (i.e., APIs) and applied our approach in each case. We adopted an *embedded design* approach for this case study by mining temporal usage patterns from multiple units of analysis (i.e., client programs). We selected three APIs (java.util, java.security, and HttpClient) that satisfied the

[2]http://www.mathworks.com/help/toolbox/stats/princomp.html

purposeful maximal sampling process (i.e., they differ enough to illustrate different perspectives on the issue investigated), that were too large to learn within a short time, and that were widely used in different client program. We selected multiple open source client programs that used at least one of the three APIs, that were active for at least 5 years, and that comprised at least 5K non-blank lines of Java code. All the client programs were developed for different domains. We applied our approach on all the client programs for each API, e.g., we identify and label phases in a client program three times, i.e., once for each API. We are still analyzing the results, but we already found examples of patterns such as the ones presented in Section II. We are also reviewing the documentation of these three APIS and we observed that these patterns could potentially be used to improve the documentation.

V. Related Work

Extensive previous work has targeted the inference of API usage patterns from a corpus of client program. Many API usage pattern inference techniques consider each function (or method) as a data set instance with attributes such as method called, fields accessed, types used, etc. Machine learning algorithms can then be applied to such data sets to infer API usage properties. As representative examples, we note the use of association rule mining by Michail [10], research on code example recommenders [2, 6], and programming rules inference tools such as PR-Miner [8]. In these approaches, the evolution of client program is not considered.

A different class of techniques analyzes the *evolution* of APIs in target systems. For example, Zimmermann et al. [15] and Ying et al. [13] developed techniques to recommend program elements that should be changed together based on the inference of association rules defined over the transactions stored in the client program's revision control system. 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 program, but without consideration for any API element actually involved in the change. In contrast, we further process the change data to automatically discover the number and type of references to API elements that are removed and added, and perform our analysis on this API usage change data.

Pattern inference techniques have also been proposed to analyze temporal aspects in API usage (for example, Acharya et al. [1] and Lo et al. [9]). 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 relations between individual elements within concepts.

VI. Conclusions

We developed a technique that analyzes the evolution of an API's integration in client programs. Our technique identifies temporal usage patterns, i.e., a sequence of usage patterns that are implemented in distinct development phases. We are currently evaluating our technique by analyzing open source client programs and investigating how their usage of three APIs evolved.

Considering its high level of automation, we believe that the initial development of our technique holds promise to help learn more about an API usage and inform both API developers and consumers. Future work will target improvements in the concept detection phase, to further ease the manual tagging effort and the extraction of reference temporal usage patterns.

Acknowledgments

The authors are grateful to Doina Precup for her advice on the use of PCA and her comments on a draft of this paper, and to Luke Bayly for his contributions to the data collection tools. This work was 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. Intl. Symp. Foundations of Soft. Eng.*, pages 25–34, 2007.

[2] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proc. 7th European Soft. Eng. Conf. and Intl. Symp. Foundations of Soft. Eng.*, pages 213–222, 2009.

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

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

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

[6] 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.

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

[8] 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.

[9] 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.

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

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

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

[13] 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.

[14] 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.

[15] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Soft. Eng.*, 31(6):429–445, 2005.