Visualization Based API Usage Patterns Refining

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

***Abstract*—Learning to use existing or new software libraries is a difficult task for software developers, which would impede their productivity. Most of existing work provided different techniques to mine API usage patterns from client programs, in order to help developers to understand and use existing libraries. However, considering only client programs to identify API usage patterns, is a strong constraint as collecting several similar client programs for an API is not a trivial task. And even if these clients are available, all the usage scenarios of the API of interest may not be covered by those clients. In this paper, we propose a visualization based approach for the refinement of Client-based Usage Patterns. We first visualize the patterns structure. Then we enrich the patterns with API methods that are semantically related to them, and thus may contribute together to the implementation of a particular functionality for potential client programs.**

I. Introduction

Software developers increasingly need to reuse functionality provided by external libraries and frameworks through their Application Programming Interfaces (APIs), where an API is the interface to implemented functionalities [1]. Hence, software developers have to cope with the complexity of existing APIs that are needed to accomplish developers' work.

Despite recent progress in API documentation and discovery [2], [3], large APIs are difficult to learn and use [1]. A large API may consist of several thousands public methods defined in hundreds of classes. Since API classes are typically meant for a wide variety of usage contexts, the elaborated documentation of an API class may be very detailed as it tries to specify all aspects that a client might need to know about the class of interest. Hence, software developers might spend considerable time and effort to identify the subset of the class's methods that are necessary and sufficient to implement their daily work. Moreover, an API method is generally used within client programs along with other methods of the API, but it is not evident to deduce the co-usage relationships between API methods from their documentations.

Therefore, identifying usage patterns for the API can help to better learn common ways to use the API, even if there exists several different ways to combine API elements (e.g., methods). In recent years, [4] much research effort has been dedicated to the identification of API usage patterns [2], [3], [5]. These existing techniques are valuable to facilitate API understanding and usage. Despite the different aspects they try to cover, these techniques are all based on client programs' code to identify common ways to use the API. In [5] we proposed 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. This technique was 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 client-based techniques are known for their accuracy. However, all the usage scenarios of the API of interest may not be covered by the clients found.

To address these issues, we propose in this paper a technique for refining Client-based Usage Patterns, premised on the analysis of API methods' dependencies within the library code itself. This technique is based on the idea that API usage patterns can be enriched with other API methods considering their mutual relationships. Our intuition is that related methods of the API may contribute together to the implementation of domain functionality in client programs. And thus may form an API usage pattern. We start from the assumptions that API methods that are semantically similar can be related to the same domain functionality, therefore these methods can be complementary in their contribution to this functionality.

The proposed approach starts with mining the Multi-Level API Usage Patterns based on the analysis of co-usage relations between the APIs methods within client programs. Then these patterns need to be refined based on the analysis of the API methods' semantic relations within the library code itself. However, refining the usage patterns is a task that needs a lot of contextual information such as the semantic relations within each usage pattern, the semantic relations between the usage patterns, the semantic relations with the unused API methods, the documentation of the API methods and the structural relations inside the methods' code. This contextual information is difficult to include in a fully-automated process. We alternatively propose a semi-automated approach with interactive visualization to perform the pattern refinement by exploring these sources of contextual information. Our contribution is mainly based on the combination of Treemap Layout algorithm [6] to visualize the hierarchical structure of the usage patterns, with the Bin Packing algorithm [7] to globally visualize the mined usage patterns.

The rest of this paper is organized as follows. Section II presents the background and the problem statement. We explain our approach in Section III. Section III-E presents the usage scenario of our visualization based refinement. Finally, Section V concludes and outlines our future work.

II. Background And Problem Statement

In this section, we introduce the necessary background, and we detail the specific issues that are addressed by our approach.

A. Multi-level API Usage Pattern

In [5] we defined a multi-level API usage pattern as a distribution, on different usage cohesion levels, of some APIs methods which are frequently used together in client programs. The rationale behind this multi-level usage patterns

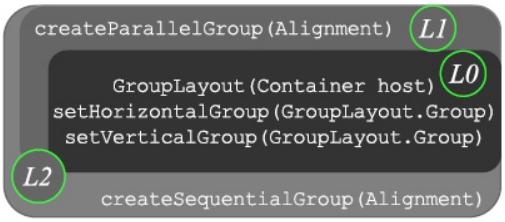

Figure 1. The cluster *L2* which represents the MLUP of class `GroupLayout`: *L0* represents the `GroupLayout`'s *core* usage pattern, then the cluster *L1*/*L2* includes *partially*/*totally* the `GroupLayout`'s *peripheral* usage pattern.

is to identify the patterns core, which represents the patterns methods that are always used together, and to reflect interfering usage scenarios of the patterns core and the rest of the APIs methods. A usage pattern includes only methods that are accessible from client programs (i.e., public methods of the API), and each usage pattern represents an exclusive subset of the APIs public methods. To illustrate the representation of multi-level usage patterns, we consider the example of the class `GroupLayout` in Swing API. `GroupLayout` is a LayoutManager that hierarchically groups components in order to position them in a Container.

After analyzing a wide variety of client methods of the `GroupLayout` class, we found that a relatively small subset of the `GroupLayouts` methods are always (consistently) used together to build layouts of Swing GUIs. Those methods are: the GroupLayouts constructor, then `setHorizontalGroup` and `setVerticalGroup` methods. Hence the core usage pattern of `Grouplayout` consists of 3 methods. Our analysis also revealed that the `Grouplayout` core usage pattern was always used with other methods of Swing. We incrementally cluster the 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 1. 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. Hence, developer can focus on the 5 methods of the usage pattern. Then, as needed, he can modify properties of their GUI layouts using other methods of the API. Our objective is precisely to refine the patterns with such methods based on contextual information.

B. Treemap Layout

One of the most used representations of a hierarchical structure by inclusion, is the treemap [6] visualization. This visualization consists of a set of rectangles where each rectangle is an element of the hierarchy, and is drawn within the limits of its parent rectangle. The parent-child relationships in the hierarchy are represented by the rectangles imbrication, which allows to efficiently use the allocated visualization

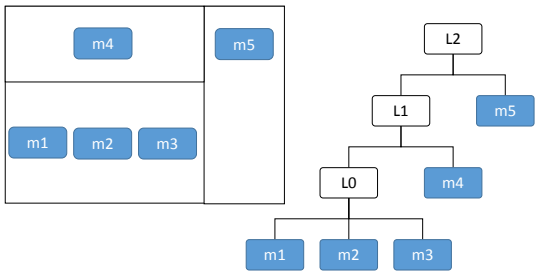

Figure 2. Treemap layout for GroupLayout pattern, m1 is the Grouplayout(Container) constructor, m2, m3, m4 and m5 are respectively the methods setHorizontalGroup(...) , setVerticalGroup (...) , createParallelGroup(...) and createSequentialGroup(...).

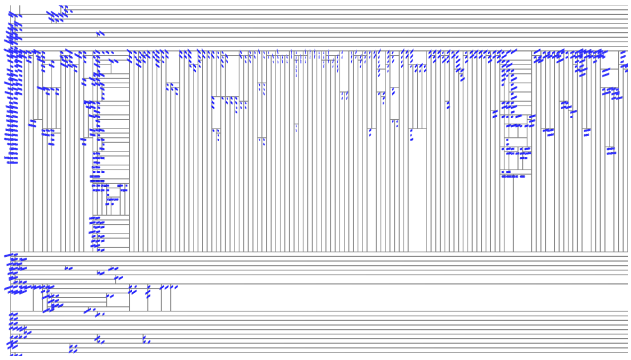

Figure 3. Standard treemap layout for the multi-level usage patterns of the Swing API.

space. Treemap layout algorithm operates on a tree structure to subdivide a rectangle into smaller regions representing the tree node and arranges the leaf of the tree inside these regions. It processes the hierarchy starting from the tree root and recursively traverses the tree. The multi-level API usage patterns are hierarchical information structures. Therefore, we can adopt the treemap layout as a visualization technique of our patterns. As we can see in Figure 2 the treemap layout of `GroupLayouts` patterns makes the structure of the hierarchy visible and intuitive.

Ideally, the co-usage relations between the usage patterns methods remain the same across all possible client methods of the usage patterns. However, APIs are open applications, and it is unfeasible to analyze all their possible usage scenarios. Hence, we need to refine them based on contextual information fetched within the library itself. To refine the usage pattern, we first need to visualize the overview of the mined patterns. At first glance and as outlined earlier, treemap layout seems to be a well-adapted technique for visualizing multi-level API usage patterns. However, in practice, multi-level usage patterns have at certain levels of the hierarchy, nodes with many children, and at other levels of the hierarchy nodes with very few children. This particular property results in a non-efficient use of the visualization space, which hinders the perception. As we can see in Figure 3, the standard treemap representation is limited because of the wasted space, which makes it difficult to visualize the overview of the mined patterns. To address this limitation, we combine Treemap Layout algorithm with the Bin Packing algorithm as described in the next section.

III. Visualization Based API Usage Patterns Refining

As outlined earlier, we define an API usage pattern as a group (i.e., cluster) of methods of the API of interest that are co-used by the API client programs.

We proposed a technique for mining Multi-Level API Usage Patterns (MLUP) to exhibit the co-usage relationships between methods of the API across interfering usage scenarios [5]. This technique was 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.

The client-based techniques are known for their accuracy. However, all the usage scenario of the API of interest may not be covered by the clients found. The challenge here is to associate the mined usage pattern to other methods that are closely related to it and can enhance its informativeness. To address this challenge, we combined program analysis with visualization techniques. The rationale behind refining the co-usage relationships based on semantic information fetched in the API code itself, is that API methods that are semantically similar can be related to the same domain functionality, and thus they can be complementary in their contribution to this functionality. Hence, those methods may be co-used together, for one particular domain purpose, in potential client programs. As a matter of fact, the domain knowledge is encapsulated in the methods vocabulary [8]. Therefore, our technique for refining API usage pattern should provide a compact visualization of the mined usage patterns and their semantics relationships.

Our approach takes as input the source code of the API to study and multiple client programs making use of this API. The output is a visualization of the mined usage patterns and their semantics relationships. To this end, we proceed as follows.

- *Extracting API methods, references and terms.* 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. We also collect terms composing the API public method name and those composing its parameter and the local variable identifiers.
- *Encoding methods' information.* Then, we compute usage and terms vectors for the API public methods. Each public method in the API is characterized by: (1) a usage vector which encodes information about its client methods, (2) a vector of the method's terms, that will be used by LSI technique [9] to construct a semantic space representation for the API of interest. The static analysis is performed using the Eclipse Java Development Tools (JDT).
- *Clustering.* Next, we use cluster analysis to group the API methods that are most frequently co-used together by client methods (multi-level usage patterns). We also use cluster analysis to identify API method groups that are semantically related.
- *Pattern visualization.* Finally, we combine Bin Packing algorithm with Treemap Layout to visualize the information mined during the clustering step.

A. Information Encoding of API Methods

In our approach, an API public method represents a point in the search space. As mentioned above, each point is represented by two vectors. The first vector is the usage vector that has constant length l, that is the number of all client methods which use the API methods. For an API method m, an entry of 1 (resp. 0) in the i^{th} position of its usage vector, denotes that m is used (resp. not used) by the i^{th} client method. The second vector is the terms' vector; it is computed from the lemmatized collected terms composing the public method name and its parameter and local variable identifiers. Similarly to usage vectors, the terms' vectors have constant length that is the number of all lemmatized collected terms composing the public methods in the API of interest. For a given API method, an entry of 1 (or 0) in the i^{th} position of the terms' vector, denotes that the i^{th} term appears (or does not appear) in the method vocabulary.

The terms' vectors of the API are used in a Latent Semantic Indexing process [9] to create a term-document matrix C. C is an $M \times N$ matrix where M is the number of all terms collected from the API and N is the number of public methods of the API. To better infer semantic similarity relations based on terms co-occurrence, a Singular Value Decomposition (SVD) [9] is applied. From the term-document matrix C, the SVD constructs three matrices: U_k is the SVD term matrix; Σ_k is the singular values' matrix; and V_k^T is the SVD document matrix, where in our case $k = min(M, N)$. The SVD document matrix, V_k^T, yields a new representation for each document (API public method), that enables us to compute document-document similarity scores in the semantic space representation as the cosine between the term vectors of the API methods.

B. Similarity and Distance Metrics

As mentioned earlier, our approach constructs clusters of API methods by grouping those that are close to each other (i.e., similar methods) either from the usage perspective or from similarity one . For this purpose, we define two similarity metrics, Usage Similarity $USim$ and Semantic Similarity $SemanticSim$.

The rationale behind the first metric is that two API methods m_i and m_j, are close to each other (i.e., similar) if they share a large subset of common client methods. $USim$, is defined in Equation (1) where $Cl\_mtd(m)$ is the set of client methods of the API method m.

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

As for the semantic similarity metric, $SemanticSim$, we use the SVD document matrix, V_k^T, as mentioned above, to compute the cosine similarity between the API methods, as defined in Equation (2). This measure is used to determine how much relevant semantic information is shared among two API methods.

$$SemanticSim(m_i, m_j) = \frac{\overrightarrow{V_i} \times \overrightarrow{V_j}}{||\overrightarrow{V_i}|| \times ||\overrightarrow{V_j}||} \quad (2)$$

Where $\overrightarrow{V_i}$ and $\overrightarrow{V_j}$ are the j^{th} and i^{th} column corresponding to m_i and m_j in the document matrix V_k^T. The semantic similarity is then normalized between 0 and 1.

Using the similarity metrics, between two API methods m_i and m_j we compute the usage distance and the semantic distance between their corresponding points p_i and p_j respectively as $UsageDist = 1 - USim(m_i, m_j)$ and $SemanticDist = 1 - SemanticSim(m_i, m_j)$.

C. Clustering algorithm

Our clustering is based on the algorithm DBSCAN [10]. 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 DBSCAN 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 algorithm to clusters only relevant points and leaves out noisy points. This explains our choice of DBSCAN for both API usage patterns mining and semantic groups identifying.

DBSCAN depends upon two parameters to perform the clustering. The first parameter is the minimum number of methods in a cluster. In our context, we set it at two, so that a cluster 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 neighbors to each other. The shorter is the distance between the methods within a cluster, the more dense is the cluster. DBSCAN, uses both $UsageDist$ and $SemanticDist$ to decide if a point belongs to the neighborhood of a given point respectively from the usage and the semantic perspective. In practice, the choice of the epsilon value is not straightforward, since we do not know before hand which density and which threshold of similarity between methods will lead to good-quality usage patterns and semantic groups. Therefore, we decided to build the clusters incrementally by relaxing the epsilon parameter, step by step. This allows relaxing the constraint on the requested density within clusters. In other words, we adapt DBSCAN algorithm for identifying usage patterns and semantic groups that may have variant densities with regard to the similarity between the clustered methods. That is to avoid one, unjustified, threshold of similarity, which may lead to less-good solutions.

D. Pattern Visualization

As mentioned earlier we opted for the treemap layout to represent our multi-level usage patterns. However, as outlined in Section II, this representation leads to a non-efficient use of the visualization space (cf. Figure 3). Alternatively, we decided to consider the visualization, as a treemap placement problem for nodes with a regular number of children and a bin packing problem for nodes with too many children. We process the hierarchy starting from the tree root and recursively traverse the tree. If the node has more than e.g., 10 children, we have to pack them into a single bin, otherwise we just have to display the treemap layout of the children. In the bin packing problem, objects of different size must be packed into a finite number of bins in a way that minimizes the number of used bins. Empirical results suggest that first fit decreasing is the best heuristic. We do not use a pure bin packing algorithm, we only have 1 bin at each target node, and we can make it as large as necessary. To pack the children nodes within a single bin, we start by placing the first (biggest) child in the top left corner, and then we split that rectangle into 2 smaller rectangles that represent the remaining white space. We do this recursively in the form of a binary tree, and we end up with a packed bin. To get the optimal size of the needed bin, we start with a small target, just big enough for the first child. And then we grow the target as needed whenever there is not enough space for the next child. We just have to grow down or right and try to keep the result roughly square. We can optionally use a border parameter to preserve between packed items.

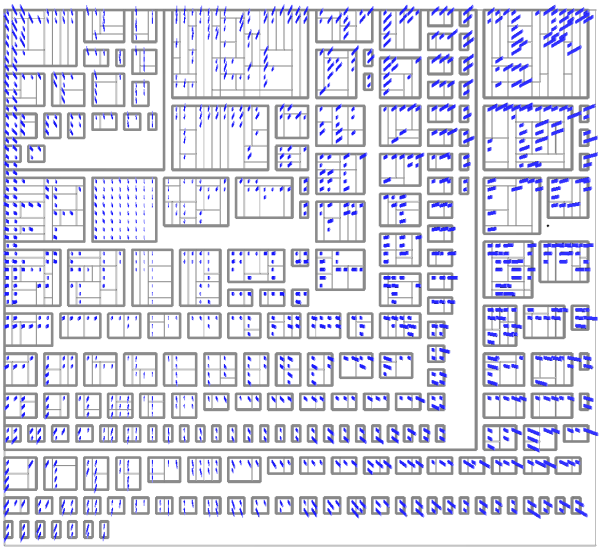

Figure 4. Combination of Bin Packing and Treemap Layout for the multi-level usage patterns of the Swing API.

Figure 4 shows that for our new placement algorithm, the empty space seems better distributed, giving it a more compact appearance. API methods are represented as 3D boxes lying on represented regions. Our visualization enables us to map different information to the graphical component of the visualization scene. For instance, method semantic group can be mapped to the boxes color, methods popularity (number of client) can be mapped to boxes height, and semantic coherence can be mapped to the region color. The rationale behind this coherence is to assess how methods within a pattern are semantically coherent. This metric is defined as the ratio of the number of elements (methods) in the pattern over the number of distinct semantic groups in that pattern. The coherence value is then normalized between [0,1]. A green color indicates high coherence within the pattern, while a red color indicates low coherence. In the next section, we show how this visualization can help to refine a usage pattern.

E. Interactive Pattern Refinement

Human intervention is needed during the API usage patterns refining task. Thus our visualization environment provides some navigation and interaction features. To refine a usage pattern, we have to start by looking at the pattern semantic coherence mapped to the region color. The choice should concern patterns presenting high coherence level, since the

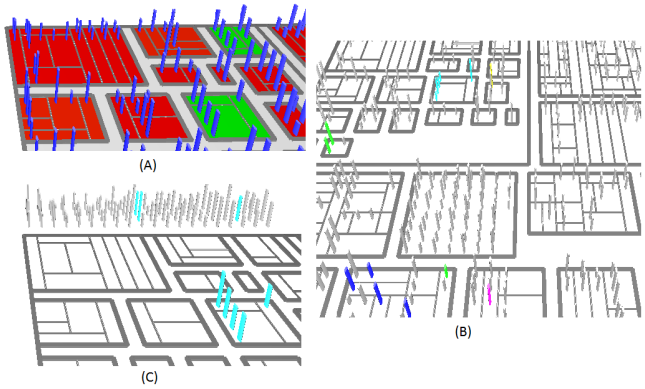

Figure 5. Usage Scenario for refining of the `GroupLayout`'s pattern of the Swing API

semantic similarity seems to be significant for these patterns, and thus, we may cover other usage scenarios of the pattern. We use a color code to represent the semantic coherence of the methods in the pattern. And we apply a color gradient on the patterns visualization, such that green color indicates high coherence values, and red color indicates low coherence values. For example, in Figure 5.(A) we can select the pattern with the green background, which is the `GroupLayout`'s pattern introduced in Section II. The next step is to check that the methods of the selected pattern do not belong to a cross-cutting semantic group, since in such cases, the semantically similar methods can be added to many other usage patterns. Here we selected a discrete set of colors to map the different semantic groups to the boxes color. In Figure 5.(B) we can see from the boxes color that methods of the `GroupLayout`'s pattern belong to a specific semantic group. The final step would be to check among the non-classified methods, whether there are those which could enrich the pattern. We can start with checking methods that have the same semantic color as the methods of the selected pattern, we can also see their documentation. In Figure 5.(C) we can see that 3 of the non-classified methods, outside the pattern's layout, have the same semantic color as the `GroupLayout`'s pattern. After looking to the methods' documentation, we added the `JLabel()` and `JTextField()` constructor to the selected patterns.

IV. Related Work

Helping developers to learn how to use an API has gained a considerable attention in recent-year research. Several directions have been investigated, in particular, understanding and learning API usage pattern [5], [11], increasing awareness of the API change impact on their usability [12]–[14], and API property visualization [13], [15]. For the sake of space, we limit our discussion to the visualization related contributions.

Parnin *et al.* [15] reported on an empirical study to investigate how Question and Answer websites, facilitate crowd documentation. The authors provided a treemap visualization tool for exploring the discussion on API elements. In their visualization, packages and classes are shown as a treemap, and color scale indicates the number of discussions on the API elements. Raula *et al.* [13] provided a visualization tool to see how the dependency relation between a system and its library evolves. They provided a system-centric and a library-centric visualization to show the successive library versions a system depends on over time. And the diffusion of dependent systems between the different versions of a library. The proposed visualization is based on the radial layout, heat-map and time-series metaphor.

In our approach we presented a new dimension of the API property visualization. We visualized the API usage pattern structure and the semantic relationship of their methods, with the aim of refining their informativeness. Our visualization is based on a combination of treemap layout and bin packing.

V. Conclusion

We developed a technique that identifies multi-level API usage patterns. Our technique allows visualization based refining of the mined pattern. We first infer groups of API methods that are highly cohesive in terms of usage by client programs. Then we detect groups of semantically related API methods. Finely we combine treemap layout and bin packing algorithms to visualize the patterns' structure and semantic relationships. In this paper, we showed that refining API usage patterns based on semantic information can lead to more informative usage patterns. In our future work, we will explore the track of new information type fetched inside the API such as structural and co-change relations. We also plan to evaluate our visualization from developers perspective.

References

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

[2] E. Moritz, M. Linares-Vásquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers, "Export: Detecting and visualizing api usages in large source code repositories," in *Int. Conf. on Automated Software Engineering (ASE)*, 2013.

[3] 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.

[4] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated api property inference techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2013.

[5] S. Mohamed Aymen, B. Omar, A. Hani, and S. Houari, "Mining multi-level api usage patterns," in *Int. Conf. on Software Analysis, Evolution, and Reengineering*. IEEE, 2015.

[6] B. Johnson and B. Shneiderman, "Tree-maps: A space-filling approach to the visualization of hierarchical information structures," in *IEEE Conference on Visualization*. IEEE, 1991, pp. 284–291.

[7] F. R. Chung, M. R. Garey, and D. S. Johnson, "On packing two-dimensional bins," *SIAM Journal on Algebraic Discrete Methods*, 1982.

[8] V. Arnaoudova, L. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, and Y. Gueheneuc, "Repent: Analyzing the nature of identifier renamings," 2014.

[9] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge university press Cambridge, 2008, vol. 1.

[10] 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.

[11] C. De Roover, R. Lammel, and E. Pek, "Multi-dimensional exploration of api usage," in *Int. Conf. on Program Comprehension*, 2013.

[12] J. Dietrich, K. Jezek, and P. Brada, "Broken promises: An empirical study into evolution problems in java programs caused by library upgrades," in *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*. IEEE, 2014.

[13] R. G. Kula, C. De Roover, D. German, T. Ishio, and K. Inoue, "Visualizing the evolution of systems and their library dependencies," in *IEEE Working Conference on Software Visualization*. IEEE, 2014.

[14] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to api deprecation?: the case of a smalltalk ecosystem," in *Int. Symposium on the Foundations of Software Engineering*. ACM, 2012.

[15] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey, "Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow," *Georgia Institute of Technology, Tech. Rep*, 2012.