API library-based identification and documentation of usage patterns

Hamzeh Eyal-Salman

To cite this version:

Hamzeh Eyal-Salman. API library-based identification and documentation of usage patterns. International Journal of Computer Applications in Technology, 2018, 58 (1), pp.63-79. 10.1504/IJCAT.2018.094065 . lirmm-03124055

HAL Id: lirmm-03124055

https://hal-lirmm.ccsd.cnrs.fr/lirmm-03124055v1

Submitted on 23 Sep 2021

API library-based identification and documentation of usage patterns

Hamzeh Eyal Salman

Mutah University,
Karak, Jordan
and
LIRMM, University of Montpellier 2,
Montpellier, France
Email: hamzehmu@mutah.edu.jo
Email: hamzehahu@gmail.com

Abstract: Application programming interfaces (APIs) are important sources for supporting source code reuse as each API provides a large set of pre-implemented functionalities that support programmers to achieve their daily work in different contexts. However, APIs provide huge number of classes and methods that hinder programmers to understand and use APIs. Numerous client-based approaches have been proposed for facilitating APIs usage through identifying frequent usage pattern. Although they represent significant efforts for helping APIs understanding, the client applications are not available for either newly released APIs libraries or APIs that are not widely used. In this paper, a non-client-based approach for frequent usage patterns identification and documentation is proposed. The approach incorporates hierarchical clustering algorithm and API's source code information. An experimental evaluation is conducted using four widely used APIs. For all studied APIs, the results show that the proposed approach is comparable with client-based approaches in terms of usage patterns cohesion.

Keywords: reuse; frequent usage pattern; API; object-oriented; understanding; documentation.

Reference to this paper should be made as follows: Eyal Salman, H. (2018) 'API library-based identification and documentation of usage patterns', *Int. J. Computer Applications in Technology*, Vol. 58, No. 1, pp.63–79.

Biographical notes: Hamzeh Eyal Salman is an Assistant Professor at Mutah University, Jordan. He obtained his PhD degree from the Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM) at University of Montpellier, France, in 2014. He got a Master in Computer Science from University of Jordon in 2010. He also has a Bachelor degree in Computer Information Systems (ranked 1st) from Al-Hussein Bin Talal University, Jordan, in 2006.

1 Introduction

Application programming interfaces (APIs) are one of the most important forms for source code reuse as they are developed only for reuse (Frakes and Kang, 2005). Each API provides a large set of functionalities (resp. their source code) that support programmers to achieve their daily work in different contexts (Kodhai and Kanmani, 2016). These pre-implemented functionalities are free of errors, and thus help to produce high quality software applications, reduce time and effort consumed in programming and testing (Roy et al., 2011). Moreover, the distinguishing feature in many frameworks and integrated development environments (IDEs) is the ability to benefit from existing APIs.

Nowadays, an API is written in object-oriented languages (such as, standard template libraries in C++ or Java SDK) and provides a large number of classes and methods. Moreover, APIs are provided by different companies where each one follows and writes in different style (Zhong et al., 2009; Bansal and Malhotra, 2016). Consequently, even experienced programmers may encounter difficulties when they use new or not widely used APIs. Furthermore, programmers struggle with identifying a set of API methods that should be invoked together to implement a specific task. Such a set is called frequent usage pattern. In fact, if API of interest is well documented, it might be not difficult for programmers to find a set of API methods that implement the task at hand. However, in most API documentations, the relationships between API methods (called co-usage relationships) are often not documented and the documentation is limited to the functionality implemented by each API method individually. Some APIs or frameworks (such as, .NET framework) documentation have sample code snippets but these snippets exhibit only one usage scenario (Zhong et al., 2009). To overcome the limitations associated with APIs documentation, programmers can use source code search engines to learn how to use API's methods. However, these search engines usually return a large number of code snippets which hinder the programmers to locate API's methods that are necessary and sufficient to implement their daily work.

Recently, there is a body of research work has been proposed for facilitating APIs usage (Robillard et al., 2013). The proposed approaches support APIs understanding through identifying sequential (Alur et al., 2005; Mandelin et al., 2005; Thummalapenta and Xie, 2007) and unordered usage patterns (Zhong et al., 2009; Saied et al., 2015b) from a given API source code. These approaches mainly rely on source code of client applications for API of interest. Although they represent a significant effort for helping APIs understanding, the client applications are not available for both new released APIs libraries and APIs that are not widely used. Also from the coverage perspective of APIs, even available client applications may not cover all usage scenarios of API of interest. As a result, client-based identification approaches of API usage patterns are only useful for identifying a subset of API methods that are used frequently by different client applications.

In this article, a non-client-based frequent usage patterns identification and documentation of API is proposed (called Non-Client-Based Usage Pattern Identification and Documentation, for short NCBUPID). The proposed approach takes as input source code of API of interest and produces as output a set of documented frequent usage patterns. Each usage pattern is documented by a set of terms describing the purpose of that pattern. NCBUPID is based on the idea that a set of API methods that always are called together and thus represent a frequent usage pattern is supposed to have strong relations. The intuition behind this idea is that API methods that collaborate and contribute to implement the same functionality are related. In this paper, the results of two types of relations between API methods and their combination are investigated. These relations are structural and textual similarities. Furthermore, NCBUPID relies on agglomerative hierarchical clustering (AHC) to cluster together a set of related API methods which represent a frequent usage pattern. It is important to mention that NCBUPID is not an alternative solution for client-based approaches. It is a solution when client applications of a given API are not available (for example, new released API). Moreover, it is expected that it does not perform better than client-based ones and the goal is to obtain results close to those achieved by client-based approaches.

In order to evaluate NCBUPID approach, a comparative evaluation is performed between NCBUPID and the most recent client-based approach in the subject called IML-FUP (Salman, 2017). Experiments have been conducted using four APIs with 89 client applications from different domains. These APIs are *HttpClient*[1], *Java Security*[2], *Swing*[3] and *AWT*[4]. In our evaluation, 30 clients were used for *AWT* and *Swing*, 17 clients were used for *Java Security* and 12 clients were used for *HttpClient*. The experimental results show that the identified usage patterns using NCBUPID remain sufficiently cohesive.

The remainder of this paper is organised as follows: Section 2 provides motivational examples. Section 3 details the proposed approach steps. Next, Sections 4 and 5 describe evaluation setting and discuss experimental results, together with the threats to validity, respectively. Then, Section 6 discusses the related work. Finally, Section 7 concludes the article and indicates future work.

2 Motivational examples

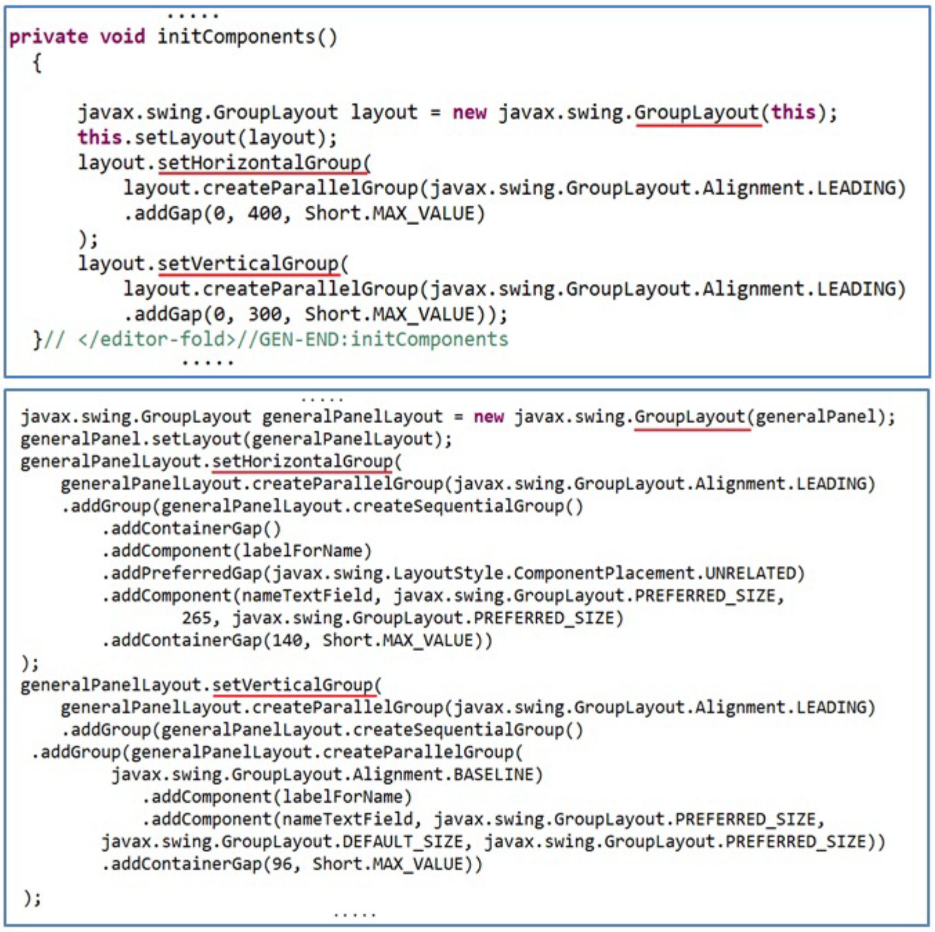
In this section, two motivational examples are presented to explain how structural and textual similarities between source code elements could be information sources for identifying co-usage relationship between API's methods. Structural similarity represents interdependencies between API's methods using method calling, parameter passing, etc. textual similarity refers to textual matching between API's methods vocabulary.

2.1 Layout design in swing API

In Swing API (Java swing api, 2016), the class *GroupLayout* is a layout manager that hierarchically groups graphical components in order to position them in a container.[5] Each group may contain any number of elements, where an element is a *Group*, *Component*, or *gap*. Usually, this class is used by every client program uses graphical user interface (GUI). Also, this class consists of 30 methods including constructor. By analysing a variety of code snippets that use *GroupLayout* class, it was found there is a subset of *GroupLayout's* methods that are always called together (Saied et al., 2015a) (see Figure 1). These methods are: *GroupLayout(Container)*, *setHorizontalGroup(Group)* and *setVerticalGroup(Group)*.

The above three mentioned methods have strong structural and textual similarities. Structurally, both *setHorizontalGroup (Group)* and *setVerticalGroup(Group)* methods use as a parameter the same object type, called *Group*. Moreover, the *GroupLayout(Container)* method calls directly both *setHorizontalGroup(Group)* and *setVerticalGroup(Group)* methods (se Figure 1). Textually, these methods share the following terms (see Figure 2): *set*, *group* and *host*.

Figure 1 Two code snippets of "*GroupLayout*" returned by Krugle code search engine



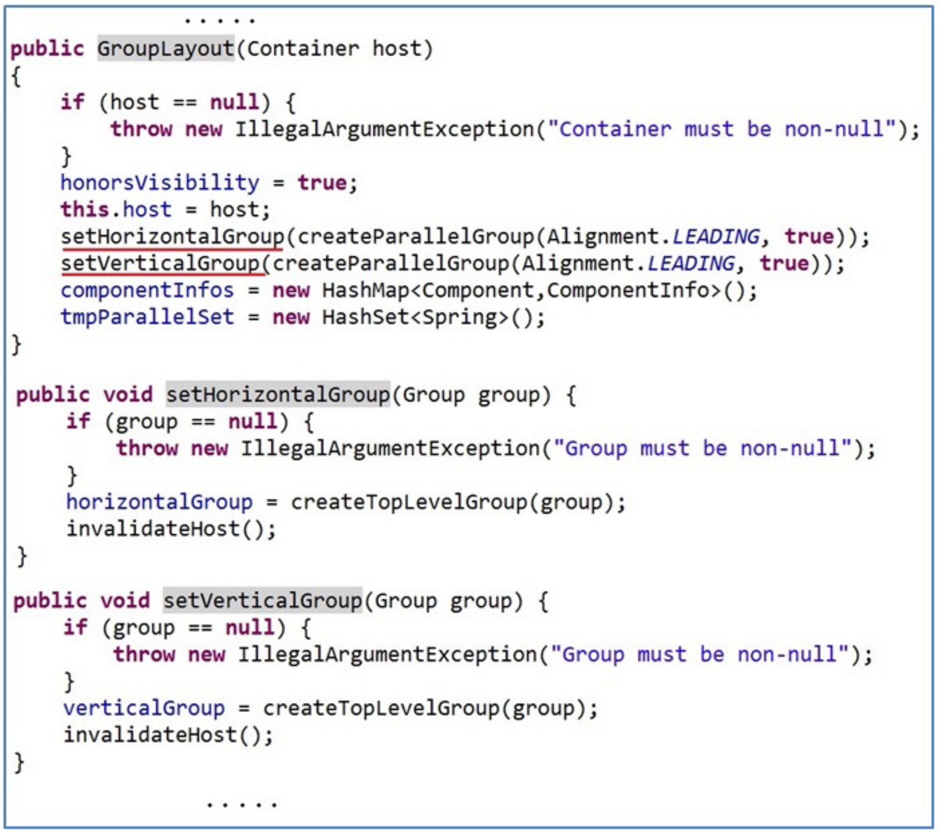
```
                .....
private void initComponents()
  {

      javax.swing.GroupLayout layout = new javax.swing.GroupLayout(this);
      this.setLayout(layout);
      layout.setHorizontalGroup(
          layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
          .addGap(0, 400, Short.MAX_VALUE)
      );
      layout.setVerticalGroup(
          layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
          .addGap(0, 300, Short.MAX_VALUE));
  }// </editor-fold>//GEN-END:initComponents
                 .....
```

```
                                 .....
javax.swing.GroupLayout generalPanelLayout = new javax.swing.GroupLayout(generalPanel);
generalPanel.setLayout(generalPanelLayout);
generalPanelLayout.setHorizontalGroup(
    generalPanelLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
    .addGroup(generalPanelLayout.createSequentialGroup()
        .addContainerGap()
        .addComponent(labelForName)
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
        .addComponent(nameTextField, javax.swing.GroupLayout.PREFERRED_SIZE,
                265, javax.swing.GroupLayout.PREFERRED_SIZE)
        .addContainerGap(140, Short.MAX_VALUE))
);
generalPanelLayout.setVerticalGroup(
    generalPanelLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
    .addGroup(generalPanelLayout.createSequentialGroup()
 .addGroup(generalPanelLayout.createParallelGroup(
         javax.swing.GroupLayout.Alignment.BASELINE)
            .addComponent(labelForName)
            .addComponent(nameTextField, javax.swing.GroupLayout.PREFERRED_SIZE,
        javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
        .addContainerGap(96, Short.MAX_VALUE))

);
                                 .....
```

Figure 2 Code segment of *GroupLayout* class



```
              .....
public GroupLayout(Container host)
{
    if (host == null) {
        throw new IllegalArgumentException("Container must be non-null");
    }
    honorsVisibility = true;
    this.host = host;
    setHorizontalGroup(createParallelGroup(Alignment.LEADING, true));
    setVerticalGroup(createParallelGroup(Alignment.LEADING, true));
    componentInfos = new HashMap<Component,ComponentInfo>();
    tmpParallelSet = new HashSet<Spring>();
}

public void setHorizontalGroup(Group group) {
    if (group == null) {
        throw new IllegalArgumentException("Group must be non-null");
    }
    horizontalGroup = createTopLevelGroup(group);
    invalidateHost();
}

public void setVerticalGroup(Group group) {
    if (group == null) {
        throw new IllegalArgumentException("Group must be non-null");
    }
    verticalGroup = createTopLevelGroup(group);
    invalidateHost();
}

                .....
```

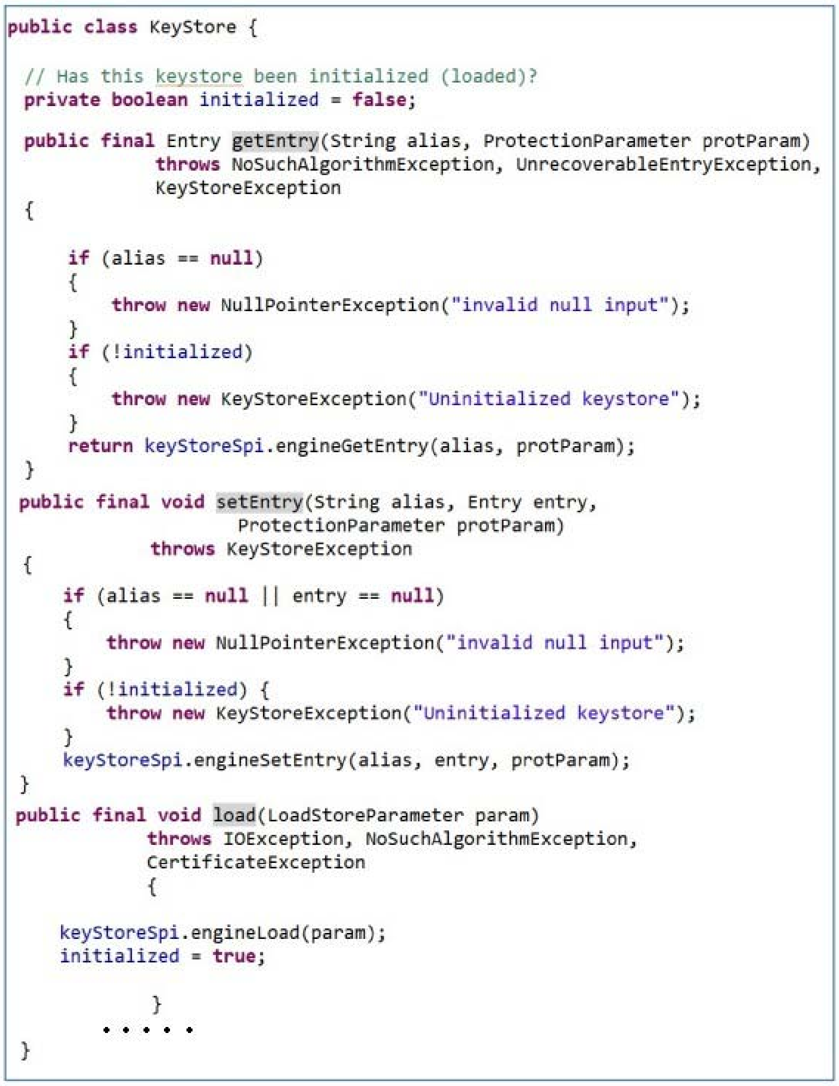
2.2 Loading and storing keys in Java security API

In Java security API (2016), a class called *KeyStore* represents a storage facility for cryptographic keys and certificates.[6] The *KeyStore* class deals with three different types of entities: *KeyStore.PrivateKeyEntry*, *KeyStore.SecretKeyEntry* and *KeyStore.TrustedCertificateEntry*. Before an instance of *KeyStore* class can be accessed, it must be loaded using the methods *load(LoadStoreParameter)*. When this instance has been loaded, it is possible to load existing entries from the *KeyStore* instance, or to store new entries into the *KeyStore* instance using the methods *getEntry(String, ProtectionParameter)* and *setEntry(String, Entry, ProtectionParameter)*, respectively.

- *load(LoadStoreParameter)*: loads this keystore object using the given *LoadStoreParameter*.[6]
- *getEntry(String, ProtectionParameter)*: gets a keystore entry for the specified alias with the specified protection parameter.[6]
- *setEntry(String, Entry, ProtectionParameter)*: saves a keystore entry under the specified alias. The protection parameter is used to protect the entry.[6]

The above-mentioned methods are identified as a frequent usage pattern in Saied et al. (2015a). These methods are structurally and textually similar. Structurally, both *getEntry*() and *setEntry*() methods use the same object type (ProtectionParameter) to protect KeyStore entries. Also, *LoadStoreParameter* object, which must be passed to the *load*() method, is used to set the *ProtectionParameter* object. Moreover, both *getEntry*() and *setEntry*() methods read the initialisation value of a field called *initialised* defined in *KeyStore* class (see Figure 3). This field is initialised by the *load*() method. Textually, the *load*(), *getEntry*() and *setEntry*() methods use the same vocabulary (see Figure 3). For example, these methods share the following terms: *entry*, *parm*, *prot* and *alias*.

Figure 3 Code segment of *KeyStore* class



```
public class KeyStore {

 // Has this keystore been initialized (loaded)?
 private boolean initialized = false;

 public final Entry getEntry(String alias, ProtectionParameter protParam)
             throws NoSuchAlgorithmException, UnrecoverableEntryException,
             KeyStoreException
 {

     if (alias == null)
     {
         throw new NullPointerException("invalid null input");
     }
     if (!initialized)
     {
         throw new KeyStoreException("Uninitialized keystore");
     }
     return keyStoreSpi.engineGetEntry(alias, protParam);
 }
 public final void setEntry(String alias, Entry entry,
                 ProtectionParameter protParam)
             throws KeyStoreException
 {
     if (alias == null || entry == null)
     {
         throw new NullPointerException("invalid null input");
     }
     if (!initialized) {
         throw new KeyStoreException("Uninitialized keystore");
     }
     keyStoreSpi.engineSetEntry(alias, entry, protParam);
 }
 public final void load(LoadStoreParameter param)
             throws IOException, NoSuchAlgorithmException,
             CertificateException
             {

     keyStoreSpi.engineLoad(param);
     initialized = true;

             }
         . . . . .
 }
```

3 The proposed approach (NCBUPID)

This section describes how the proposed approach identifies non-clients-based frequent usage patterns from APIs source code using hierarchical agglomerative clustering.

3.1 Overview

Figure 4 presents an overview of frequent usage patterns identification process which defines five steps. The first step takes as input an API source code which is statically parsed to identify public methods. In the second step, each public method is characterised by *interdependency* and *term* vectors which encode structural and textual information associated with that method, respectively. Then in the third step, similarities between these public methods (resp. their vectors) are computed. The proposed process relies on the following heuristics to compute such similarities:

- *Heuristic 1* [*structural similarity*]: it refers to interdependencies among public methods (e.g. method calls, parameters passing, etc.) where methods that depend on each other are expected to collaborate in order to implement the same domain task or functionality.
- *Heuristic 2* [*textual similarity*]: it refers to textual matching between terms derived from identifiers of API's public methods. Identifier names record important domain knowledge which represent functionality(s) implemented by these code elements (identifiers). Therefore, when two or more methods share a lot of terms, it is expected these methods contribute to implement the same domain task or functionality especially when the developers use the same vocabulary across source code elements.
- *Heuristic 3* [*combining structural and textual similarities*]: it refers to the integration between both structural and textual similarities. The idea behind such integration is that by combining these sources of information their drawbacks can be minimised and better results can be achieved.

The fourth step in the proposed process clusters similar methods together using agglomerative hierarchical clustering algorithm (AHC) as each FUP is a group of public methods which are used together. Each resulted cluster represents a frequent usage pattern. Finally, each identified pattern is documented automatically by finding keywords describe the purpose of that pattern. In the following sections, each process step is detailed.

3.2 Parsing API's public methods source code

This is the first step in the identification process where source code of each given API is statically analysed. This static analysis allows finding public methods of that API and then identifies interdependencies between these methods and extract textual information. Such static analysis is performed using the Eclipse Java Development Tool (JDT).

In object-oriented technology, there are numerous interdependency relationships between source code elements. This study investigates the results of a set of these relationships which are recommended by motivation examples. Such interdependencies represent structural similarity which includes:

1 Calling the same method(s): when the same API method is called by two or more API methods.

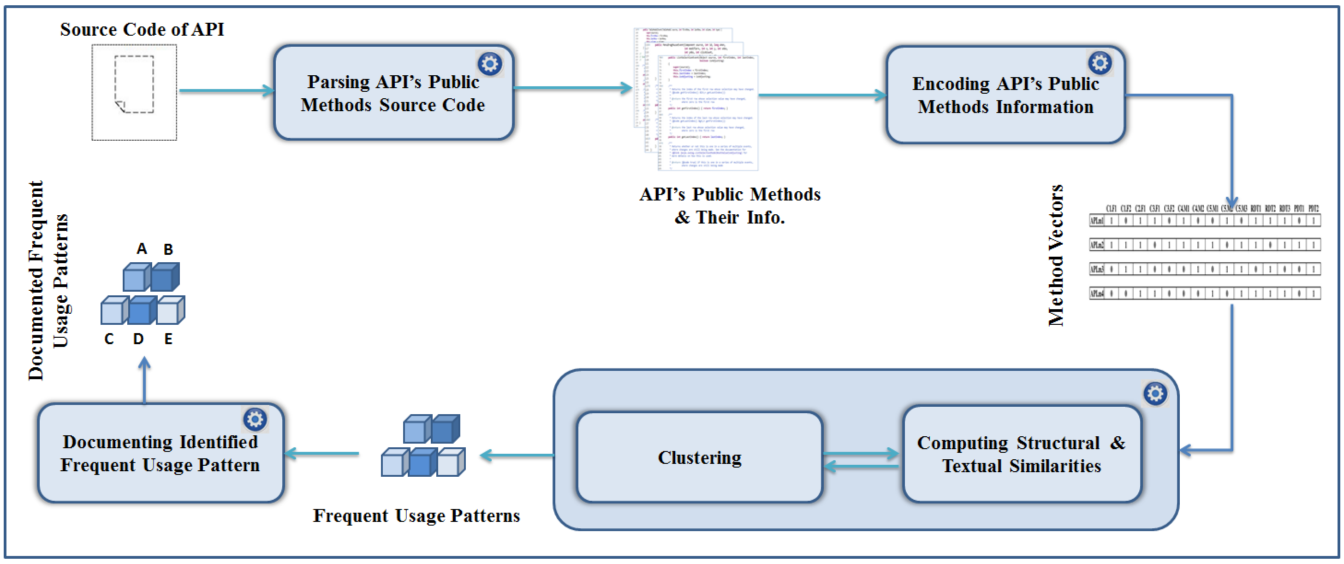
2 Accessing the same attribute(s): when the same attribute(s) is accessed by two or more API methods.

3 Using the same parameter type(s): when the same parameter type(s) is used by two or more API methods.

4 Returning the same value type: when the returned-value type is the same for two or more API methods.

To find textual links between API methods, textual source code information specific to each API method is only considered in this study. This information includes *method name*, *parameter names* and *local variable names*.

Figure 4 The proposed approach overview


Source Code of API
Parsing API's Public Methods Source Code
API's Public Methods & Their Info.
Encoding API's Public Methods Information
Method Vectors
Computing Structural & Textual Similarities
Clustering
Frequent Usage Patterns
Documenting Identified Frequent Usage Pattern
Documented Frequent Usage Patterns
A
B
C
D
E

3.3 Encoding API's public methods information

In this proposed approach, clustering algorithm starts from a set of points so that each point represents an API public method. Therefore, source code information (i.e., both structural and textual information) associated with each public method should be encoded as a point in clustering algorithm search space. To do this, two vectors are created for each API public method: *interdependency* and *term* vectors.

The interdependency vector of each API public method has a constant length that represents the number of all interdependency relationships existing between API's public methods. Figure 5 shows a toy example of API consisting of four public methods with 15 interdependency relationships. Therefore, each API method will have interdependency's vector of length 15. For a given API method, an entry 1 or 0 in the *i*-th position denotes that *i*-th element (either field, method or data type) is referenced or not referenced respectively by that API method.

Similarly to interdependency vector, the term vector of each API public method has constant length that represents number of all terms composing public method names, parameter names and local variable names of a given API. Figure 6 shows a toy example of term vector representations of API consisting of four public methods. For each public method, an entry in the *i*-th position refers to term frequency in that public method.

3.4 Computing structural and textual similarities

In order to group and aggregate together similar API public methods into clusters which represent usage patterns, a similarity metric is needed. In this study two similarity metrics are defined: *structuralSim* and *textualSim*. The *structuralSim* is used to capture interdependencies relationships between API public methods while the *textualSim* is used to compute textual matching between API public methods vocabulary.

The *structuralSim* between two API public methods is defined in equation (1) using Jaccard similarity coefficient. For two given API public methods (m_i and m_j), the rationale behind using the Jaccard similarity is that two API methods are close to each other if they share a large subset of the called methods, fields, 160 returned data types and parameter data types in their corresponding interdependency vectors.

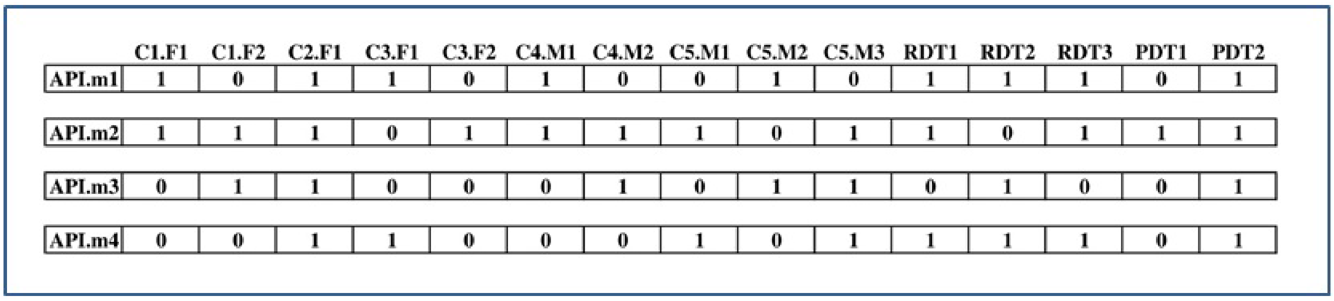
$$structuralSim\left(m_i, m_j\right) = \frac{\left| \begin{array}{l} interdependencyVec\left(m_i\right) \\ \cap interdependencyVec\left(m_j\right) \end{array} \right|}{\left| \begin{array}{l} interdependencyVec\left(m_i\right) \\ \cup interdependencyVec\left(m_j\right) \end{array} \right|} \quad (1)$$

where *interdependencyVec* denotes to interdependency vector of m_i and m_j.

The *textualSim* between two API public methods is defined in equation (2) using cosine similarity (Saied et al., 2015a; Marcus and Maletic, 2003; Kayarvizhy et al., 2016). For two given API public methods, this metric is used to determine how much relevant textual information is shared among their corresponding term vectors.

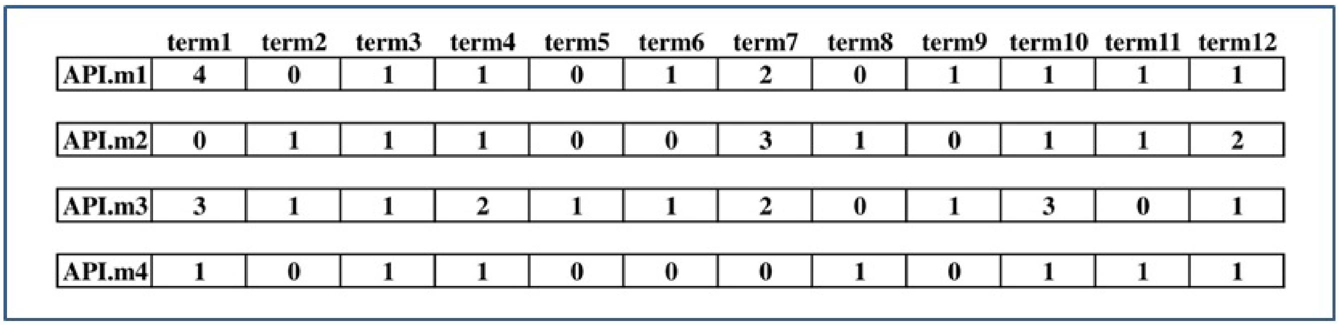
$$textualSim\left(m_i, m_j\right) = \frac{\overrightarrow{m_i} \cdot \overrightarrow{m_j}}{\left\|\overrightarrow{m_i}\right\| \left\|\overrightarrow{m_j}\right\|} \quad (2)$$

Figure 5 An graphical representation of four interdependency vectors consisting of CM (Class.Method), (Class.Field), RDT (returned data type) and PDT (parameter data type)



	C1.F1	C1.F2	C2.F1	C3.F1	C3.F2	C4.M1	C4.M2	C5.M1	C5.M2	C5.M3	RDT1	RDT2	RDT3	PDT1	PDT2
API.m1	1	0	1	1	0	1	0	0	1	0	1	1	1	0	1
API.m2	1	1	1	0	1	1	1	1	0	1	1	0	1	1	1
API.m3	0	1	1	0	0	0	1	0	1	1	0	1	0	0	1
API.m4	0	0	1	1	0	0	0	1	0	1	1	1	1	0	1

Figure 6 An graphical representation of term vector consisting of methods vocabulary



	term1	term2	term3	term4	term5	term6	term7	term8	term9	term10	term11	term12
API.m1	4	0	1	1	0	1	2	0	1	1	1	1
API.m2	0	1	1	1	0	0	3	1	0	1	1	2
API.m3	3	1	1	2	1	1	2	0	1	3	0	1
API.m4	1	0	1	1	0	0	0	1	0	1	1	1

In order to compute the similarity resulted by combining structural and textual heuristics, the *structuralSim* and *textualSim* are combined in equation (3). For two given API methods m_i and m_j, the combined similarity (*combinedSim*) is computed as follows:

$$combinedSim\left(m_i, m_j\right) = \frac{structuralSim\left(m_i, m_j\right) + textualSim\left(m_i, m_j\right)}{2} \qquad (3)$$

3.5 *Clustering algorithm*

To identify groups of public methods which represent usage patterns, an algorithm should be used. Among the possible algorithms, a clustering algorithm is opted. This kind of algorithms is used to group elements using similarity function. This makes it suitable for the problem addressed in this study as similarity metrics defined prior will play the role of a similarity function.

Clustering algorithms are classified into hierarchical or non-hierarchical. Hierarchical clustering algorithms are further categorised into agglomerative (AHC for short) and divisive. In this study, AHC is opted to cluster similar API public methods into frequent usage patterns. AHC starts with singleton clusters (i.e. clusters having only one object) and recursively merges the two most similar clusters in each stage. These singleton clusters initially consist of individual API public methods and later of clusters of public methods formed during the previous stages. Based on this description of AHC, it can be deduced that AHC computes similarity among public methods, among clusters, and between clusters and public methods. In this study, the application of AHC relies on the following two steps.

3.5.1 *Building a hierarchy of clusters*

For a given set of API public methods, AHC groups similar methods into clusters. The basis for such clustering is the strength of the relationship between them. This relationship refers to structural similarity, textual similarity and a combination thereof.

AHC works by creating a tree of nested clusters, called a *dendrogram*. A dendrogram is a tree representation frequently used to illustrate the arrangement of the clusters produced by hierarchical clustering (Haifeng and Zijie, 2010). AHC is a adapted to build a dendrogram from a given set of API public methods according to Algorithm 1. This algorithm relies on a series of successive binary mergers, initially of individual methods and later of clusters formed during the previous stages. In the beginning, it puts each method in its own cluster. Among all current clusters, the two most similar clusters (mostSimilarClusters()) are picked. Then, these two clusters are replaced with a new cluster by merging the two original ones. The process continues until only one cluster remains such that at each iteration only one pair of clusters that have the highest relationship strengths are merged. This single cluster represents a dendrogram (dendgr) that contains a set of nested clusters. Merging two clusters mean aggregation the method vectors of these clusters using the logical disjunction in one vector (see lines 11–14). Also, it is important to mention that the function *mostSimilarClusters*() represents structural similarity, textual similarity or their combination depending on the type of similarity to be investigated.

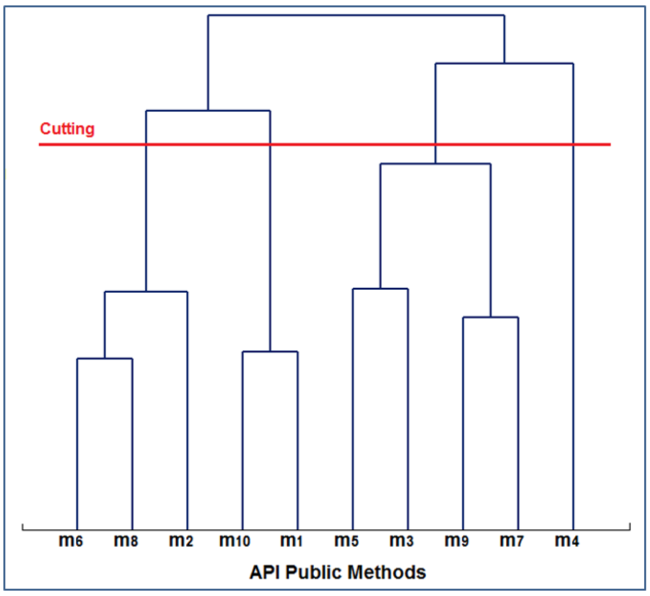
Algorithm 1: BuildingDendrogram

```
   Input: $methods$ // API public methods
   Output: $dendgr$ // Dendrogram Tree
1  $stack\ clusters \leftarrow methods$
2  while ($|clusters| > 1$) do
3      $(Clu1, Clu2) \leftarrow mostSimilarClusters(clusters)$
4      $Pop(Clu1, clusters)$
5      $Pop(Clu2, clusters)$
6      $Clu3 \leftarrow Merge(Clu1, Clu2)$
7      $Push(Clu3, clusters)$
8  end
9  $dendgr \leftarrow get(clusters)$
10 return $dendgr$

11 Function $Merge$ ($cluster1$, $cluster2$)
12 {
13 return ($cluster1$ OR $cluster2$)
14 }
```

Figure 7 shows an example of dendrogram tree. At the lowest level, each method is in its own cluster. At the highest level, all methods belong to the same cluster. The internal nodes represent new clusters formed by merging the clusters that appear as their children in the tree.

Figure 7 An example of a dendrogram tree



3.5.2 *Selecting candidate frequent usage patterns*

Breaking the generated dendrogram tree based on predefined criteria allows grouping classes into clusters. Each resulting cluster can be a candidate frequent usage pattern. Therefore, the appropriate breaking points should be selected to obtain frequent usage patterns. This selection is performed by an algorithm based on a depth-first search (refer to Algorithm 2). This algorithm takes as input the dendrogarm tree and returns a set of clusters. These clusters are interpreted as frequent usage patterns. This algorithm starts by comparing the similarity value (Sim()) of each node in the dendrogram (starting from the root) and its sons. If the similarity value of the focused node is less than the average of the similarity values of its two sons, then the algorithm continues to the next son nodes. Otherwise, the focused node is identified as a frequent usage pattern, added to the FUPs accumulator and the algorithm computes the next node in the stack (*traversedClusters*). In this way, the most relevant frequent usage patterns will be identified as the traversal continues.

To visualise how Algorithm 2 selects clusters (i.e., frequent usage patterns), see Figure 7. The red horizontal line determines the cutting points. Based on these points, four clusters can be obtained as follows. A first cluster contains methods m6, m8 and m2. A second cluster contains methods m10 and m1 while methods m5, m3, m9 and m7 belong to a third cluster. Finally, only one method m4 forms a fourth cluster.

Algorithm 2: Selecting Candidate Frequent Usage Patterns

```
Input: $Dendrogram(dendgr)$
Output: FUPs // FUPs: Frequent Usage Patterns
1  $stack\ traversedClusters$
2  $traversedClusters.push(root(dendgr))$
3  while ($|traversedClusters| > 0$) do
4      $parent \leftarrow traversedClusters.pop(traversedClusters)$
5      $son1 \leftarrow getSon1Cluster(parent, dendgr)$
6      $son2 \leftarrow getSon2Cluster(parent, dendgr)$
7      $avg \leftarrow average(Sim(son1), Sim(son2))$
8      if ($Sim(parent) > avg$) then
9          $add(parent, FUPs)$
10     else
11         $traversedClusters.push(son1)$
12         $traversedClusters.push(son2)$
13     end
14 end
15 return $FUPs$
```

3.6 Documenting identified usage patterns

Frequent usage patterns can be efficiently used if their documentation (e.g., main purpose, name, etc.) is available. Thus, the need to document the identified usage patterns is necessary. To achieve this goal, a heuristic is used to discover the purpose of an identified usage pattern. In many object-oriented languages, method names are sequences of terms concatenated using a camel-case notation (e.g., *setLeftComponent*(), *getHighlightInnerColor*() and *closeMenu*()). The first term of a method name refers to the functionality to be performed by that method (*set*, *get* and *close*). The other terms indicate to objects or input which are associated with that functionality (*LeftComponent*, *Menu* and *HighlightInnerColor*). According to these assertions, each identified usage pattern is documented using the following steps: *decomposing method names*, *token frequency computing* and *constructing the pattern name*.

3.6.1 Decomposing method names

For a given frequent usage pattern, method names of that pattern are split into tokens according to the camel-case convention. In this convention the uppercase case letters and underscore are used as delimiters for splitting. For example *setLeftComponent*() is split into *set*, *left* and *component*. However, it is possible to encounter single case method term (such as, maxvalues), abbreviations and acronyms. To handle such name compositions, an algorithm proposed by Warintarawej et al. (2015) is used.

3.6.2 Token frequency computing

In this step, the tokens extracted from method names of each usage pattern undergo a preprocessing step. A preprocessing involves normalising the tokens such as stop word removal. Then, a token frequency is computed and assigned to each token extracted from a method name. For a given token, this frequency indicates to the number of times a token is used for naming API public methods of a given frequent usage pattern.

3.6.3 Constructing the pattern name

In this step, a usage pattern name is constructed based on the high frequency tokens. The first word of the pattern name is the first high frequency token. The second word of the pattern name is the second high frequency token and so on. The number of words used in the pattern name is specified by the user. When many tokens have the same frequency, all the possible combinations are given to the user and he can select the appropriate one.

4 Evaluation setting

This section describes the setting of experiments. Particularly, research questions, studied APIs, a collection of client applications and used metrics for evaluating the effectiveness of the proposed approach are defined.

4.1 Research questions and evaluation metrics

The main goal of this study is to evaluate whether the proposed approach can identify API's cohesive usage patterns that are comparable to those identified using clients-based approaches. Therefore, the following research questions are formulated.

- RQ1 [Pattern Usage Cohesion]: To which extent the textual and structural heuristics individually and a combination thereof help identifying API's methods that always are invoked together?
- RQ2 [Comparable Usage Pattern]: To which extent the identified patterns are comparable to those identified by client-based approaches?

In order to address the first research question (RQ1), the impact of structural and textual heuristics on identifying usage pattern is studied. Therefore, results of these heuristics separately and in combination are investigated for identifying usage patterns from selected APIs. To evaluate the results of studied heuristics and their combination in terms of the cohesion of identified patterns, a measure for a pattern usage cohesion is needed. Such a measure evaluate whether an identified pattern is cohesive enough to exhibit co-usage relationships between the API methods from the perspective of API client applications. For this, Service Interface Usage Cohesion metric (SIUC) is used (Perepletchikov et al., 2010). A service in this metric is deemed to be externally cohesive when all of its service operations are invoked by all the clients of this service. This definition for service cohesion is similar to pattern usage cohesion. Therefore SIUC metric was adopted in Saied et al. (2015b) and Salman (2017) and referred to as Pattern Usage Cohesion (PUC) and Multi-Level Pattern Usage Cohesion (MLPUC), respectively. PUC values take a range in [0…1]. The larger the value of PUC is, the better the usage cohesion. The ideal usage cohesion occurs when PUC is equal to 1. This means that all the pattern's methods are actually always used together. For a given pattern *p*, PUC is defined as follows (Saied et al., 2015b):

$$PUC(p) = \frac{\sum_{client} \frac{|PMe(client)|}{|PMe|}}{|clients(p)|} \tag{4}$$

where $|clients(p)|$ is the total number of all client methods of the API's methods in p. $|PMe|$ is the number of all API's methods in p. $|PMe(client)|$ is the number of API's methods in p invoked by a client method client.

In order to address the second research question (RQ2), NCBUPID is compared to the most recent client-based frequent usage patterns identification approach in the subject, called IML-FUP (Salman, 2017). To identify API usage patterns, IML-FUP uses formal concept analysis (FCA) technique to cluster API's methods which always or frequently are used together within a variety of client applications of the studied API. This comparison is performed in terms of average PUC, average number and size of identified patterns. For fair comparison, the proposed heuristics are evaluated using the same set of API methods that were used by IML-FUP (i.e., a set of API methods called by the client applications considered in this study).

4.2 Experimental setup

To assess the first research question (RQ1), NCBUPID is run three times on each studied API. In the first and second runs, structural and textual heuristics are considered separately, respectively. In the third run, the combination of these heuristics is considered. For each run, API's public methods only are used as a data set to be clustered. For each studied API, identified API usage patterns for three runs are collected and analysed. It is important to mention that some of identified patterns by NCBUPID are not covered (called) by selected client applications – in spite of these clients represent a large number of applications from different domains. Therefore for each selected API, only API's methods which are covered by client applications can be an input for NCBUPID.

To assess the second research question (RQ2), all APIs and their client applications used by IML-FUP are considered in this study. Tables 1, 2, 3 and 4 show these APIs and applications. API's methods which only are called by these client applications can be clustered by IML-FUP. Therefore, for fair comparison between NCBUPID and IML-FUP the set of API's methods called by client applications of each API is identified. Then, this set is used as an input for both NCBUPID and IML-FUP. Then, identified patterns from these approaches are evaluated using the evaluation measures (PUC, average number and size of identified pattern).

Table 1 Client applications corresponding to SWING API

API	*Client applications*	*Description*
swing	LaTeXDraw2.0.8	Is a graphical drawing editor for LaTeX
	SweetHome3D-3.4	An interior design application
	RapidMiner	An integrated environment for machine learning and data mining
	Msproject	MS-Project import/export plugin for GanttProject
	Pert	The PERT plugin for GanttProject
	Mogwai	Java 2D and 3D visual entity relationship design and modelling (ERD,S QL)
	G4P (GUI for processing)	A library that provides a rich collection of 2D GUI controls
	Apache-jmeter-2.11	Java application designed to test and measure performance
	Art-of-Illusion	A 3D modelling and rendering studio
	AtlasCreator	An application creates off-line atlases of raster maps for various cell phone applications
	Code2uml	A tool for constructing UML class diagrams from java .class and .jar files
	Davmail	A POP/IMAP/SMTP/Caldav/Carddav/LDAP gateway allowing users to use any mail client
	EasyFileShare	An application to share files from your PC to any other device
	Freemind	A mind-mapping editor
	GanttProject core	An application for project management and scheduling
	GLIPS	A cross-platform SVG graphics editor

Table 1 Client applications corresponding to SWING API (continued)

API	*Client applications*	*Description*
	Java-chat	An application for chatting
	JEdit	A text editor
	JHotDraw	A Java GUI framework for technical and structured Graphics
	Mailcarbon	An application for backup emails from one server to another over IMAP
	Neuroph	A lightweight Java Neural Network Framework
	Metawidget	A smart widget Building User Interfaces for domain objects
	VASSAL-3.2.15	An engine for building and playing human-vs.-human games
	Open-so-frontend	An application for managing the Stack Overflow family of sites
	Swingx	Contains extensions to the Swing GUI toolkit
	Paros	A Java based HTTP/HTTPS proxy for assessing web application vulnerability
	PlotDigitizer	An application to digitise data points off of scanned plots and scaled drawings
	Pmd	A source code analyser
	RESTEasy	A JBoss project that provides various frameworks to build RESTful Web Services
	xsmile	A Java based XML browser

Table 2 Client applications corresponding to AWT API

API	*Client applications*	*Description*
awt	LaTeXDraw2.0.8	Is a graphical drawing editor for LaTeX
	SweetHome3D-3.4	An interior design application
	RapidMiner	An integrated environment for machine learning and data mining
	Msproject	MS-Project import/export plugin for GanttProject
	Pert	The PERT plugin for GanttProject
	Mogwai	Java 2D and 3D visual entity relationship design and modelling (ERD, SQL)
	G4P (GUI for processing)	A library that provides a rich collection of 2D GUI controls
	Apache-jmeter-2.11	A project that can be used as a load testing and measure performance tool
	Art-of-Illusion	A 3D modelling and rendering studio
	AtlasCreator	An application creates off-line atlases of raster maps for various cell phone applications
	OpenLaszlo	An open source platform for the development and delivery of rich Internet applications
	Davmail	A POP/IMAP/SMTP/Caldav/Carddav/LDAP gateway allowing users to use any mail client
	EasyFileShare	An application to share files from your PC to any other device
	Freemind	A mind-mapping editor
	GanttProject core	An application for project management and scheduling
	GLIPS	A cross-platform SVG graphics editor
	Java-chat	An application for chatting
	JEdit	A text editor
	JHotDraw	A Java GUI framework for technical and structured Graphics
	Mailcarbon	An application for backup emails from one server to another over IMAP
	Neuroph	A lightweight Java Neural Network Framework
	Metawidget	A smart widget Building User Interfaces for domain objects
	VASSAL-3.2.15	An engine for building and playing human-vs.-human games
	Open-so-frontend	An application for managing the Stack Overflow family of sites
	Swingx	Contains extensions to the Swing GUI toolkit
	Paros	A Java based HTTP/HTTPS proxy for assessing web application vulnerability
	Htmlpdf	The html and pdf export plugin for GanttProject
	Pmd	A source code analyser
	RESTEasy	A JBoss project that provides various frameworks to build RESTful Web Services
	xsmile	A Java based XML browser

Table 3 Client applications corresponding to SECURITY API

API	*Client applications*	*Description*
	YaHPConverter	A Java library that allows you to convert an HTML document into a PDF document
	ApacheJackrabbit	Is an open source content repository for the Java platform
	Apache-jmeter	A project that can be used as a load testing and measure performance tool
	Davmail	A POP/IMAP/SMTP/Caldav/Carddav/LDAP gateway allowing users to use any mail client
	Heritrix	A web crawler
	Hibernate	An Object/Relational Mapper tool
	HttpclientAuthHelper	An application to authenticate Httpclient with services that use NTLM, KERBEROS and SSL
	Lcrypto	Bouncy Castle Cryptography
security	MinaSource	An application framework which develop high performance and high scalability network applications
	Mule-3.x	A lightweight enterprise service bus (ESB) and integration framework
	OpenLaszlo	An open source platform for the development and delivery of rich Internet applications
	Paros	A Java based HTTP/HTTPS proxy for assessing web application vulnerability
	RESTEasy	A JBoss project that provides various frameworks to build RESTful Web Services
	RSSOwl	An aggregator for RSS and Atom News feeds
	wildfly	An application server
	Xsmile	A Java based XML browser
	Xstream	An application to serialise objects to XML and back again

Table 4 Client applications corresponding to HTTPCLIENT API

API	*Client applications*	*Description*
	Davmail	A POP/IMAP/SMTP/Caldav/Carddav/LDAP gateway allowing users to use any mail client
	Heritrix	A web crawler
	HttpclientAuthHelper	An application to authenticate Httpclient with services that use NTLM, KERBEROS and SSL
	HueMorseCommunicator	An application to sends messages through a Philips Hue light using morse code
	Javabook-client	A Java API for Facebook
	Mule-3.x	A lightweight enterprise service bus (ESB) and integration framework
httpclient	OpenLaszlo	An open source platform for the development and delivery of rich Internet
	Paros	A Java based HTTP/HTTPS proxy for assessing web application vulnerability
	Rabbitmq-management	Java project to handle rabbitmq administration
	RSSOwl	An aggregator for RSS and Atom News feeds
	Sage-gateway	An application for data archival, preservation and access for all projects of NSF's Arctic Science Program
	Weibo4j-oauth2	Sina Mblog openAPI javaSDK

4.3 Data collection

To execute the empirical evaluation of NCBUPID, a large group of 89 client applications are used. This group is open-source Java projects from different domains and sizes[7] and they are developed using different Java APIs. Among these APIs, four widely used API libraries are considered also for empirical evaluation. These APIs are: *httpclient*, *security*, *swing* and *awt*. The *httpclient* API is used to facilitate communication over web services. The *security* API provides security framework. The *awt* API help designers to create interfaces and paint images and graphics. Finally, the *swing* API concerns with GUI. Tables 1, 2, 3 and 4 present descriptive information for clients applications developed using these APIs.

The client applications and their APIs are used as follows: 30 client applications are chosen for *swing* and *awt* APIs, two groups of 17 and 12 clients are chosen for *security* and *httpclient* APIs, respectively (see Tables 1, 2, 3 and 4).

5 Experimental results analysis

This section presents and discusses experimental results of NCBUPID and answers of the research questions defined in section 4.1

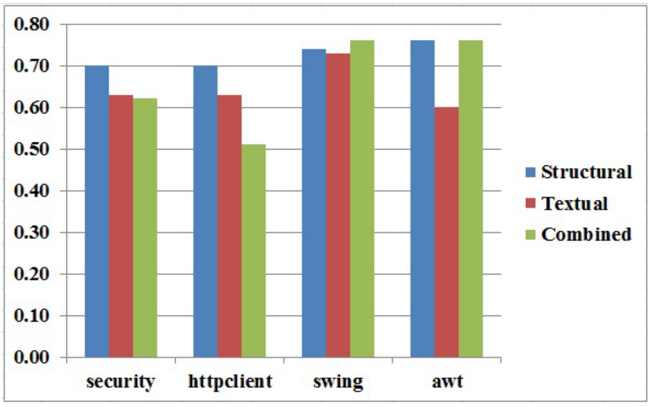
Table 5 Statistics on source code of APIs of interest

API Name	*#classes & Interfaces*	*#Public Methods*
swing	2570	18,515
httpclient	871	7290
security	688	5764
awt	491	4778

5.1 *Impact of the proposed heuristics (RQ1)*

In order to answer the first research question (RQ1), the usage cohesion of identified patterns is analysed. Figure 8 shows average cohesion values of patterns identified from each API by applying separately and in combination the structural and textual heuristics. The results of applying each heuristic are analysed in the following subsections.

Figure 8 Average cohesion values of patterns identified using the investigated heuristics



5.1.1 *Analysing the structural heuristic results*

As shown in Figure 8, it is clear for all studied APIs when only structural interdependencies between API's methods are used, the identified patterns have strong co-usage relations between the patterns' methods. In fact, the average usage cohesion values of these patterns take a range between 0.70 for security API and 0.76 for awt API.

5.1.2 *Analysing the textual heuristic results*

As shown in Figure 8, for all APIs when only textual similarity is used between API methods, the co-usage relationships between API's methods of identified patterns are slightly degraded comparing with structural similarity. Indeed using this heuristic alone, average cohesion values are in a range between 0.60 for awt API and 0.73 for swing API.

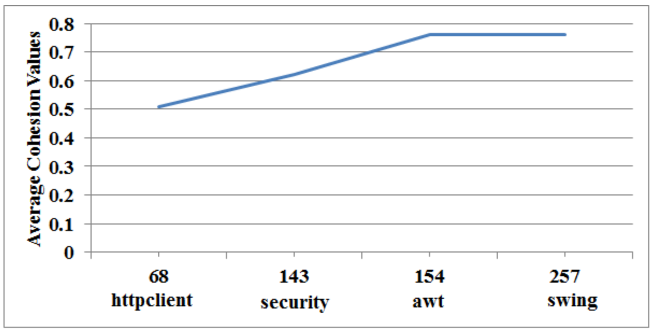
5.1.3 *Analysing results of combining structural and textual heuristics*

As seen in Figure 8 when a combination of structural and textual heuristics is applied, usage patterns identified from APIs that have a large number of clients (such as, awt and swing) have cohesive usage relationships among their API's methods like or better than structural heuristic. Indeed, average cohesion values of the patterns decrease when number of clients of API decreases. Average cohesion values of patterns identified from awt and swing that have 30 client applications are equal to 0.76 while for security and httpclient that have 17 and 12 client applications are 0.62 and 0.52, respectively.

The results presented in Figure 8 show that structural heuristic helps to identify usage patterns having strong co-usage relations between their API's methods than those patterns identified using textual heuristic alone for all studied APIs. This is due to two reasons. On one hand, structural source code information represents strong links between source code elements that collaborate to implement specific functionality or similar functionalities. On the other hand, textual heuristic mainly depends on vocabulary used by developers to write source code statements and comments, and size of textual source code information considered. When developers use different vocabularies, this leads to slight degradation in pattern cohesion comparing to structural information. Moreover, when number of client applications of API is small (i.e., number of public methods covered in that API) this may negatively impact pattern cohesion because there is no enough textual source code information for matching.

However, when multiple source of information are combined (i.e., structural and textual heuristics in combination), often this yields better results than if these sources are used individually. The sources of information have their individual benefits and drawbacks, but when they are combined, those drawbacks can be minimised and better results can be obtained. It is clear from Figure 8 that average cohesion value by applying combined heuristic on swing API is better than applying each heuristic individually. Also this is true for awt API where the result of combined heuristic is better than the textual heuristic and equals to structural heuristic. However, for security and httpclient APIs the average cohesion values of the combined heuristic are less than average values resulted by applying each heuristic individually. This is due to that number of covered methods in security and httpclient APIs are small where the numbers of client applications of these APIs are 17 and 12, respectively. Figure 9 shows the relationship between the average cohesion values of combined heuristic and number of covered methods for each studied API.

Figure 9 Relationship between average cohesion of combined heuristic and number of covered methods of each API



As a summary, the structural heuristic helps to identify co-usage relationships between the API's methods with high precision and always has better contribution than textual heuristic for identifying cohesive usage patterns across all studied APIs. On average 70% and up to 76% of API's methods in an identified usage pattern using structural heuristic are always uniformly co-used together. Combining the structural and textual heuristics perform the best for identifying co-usage relations between API's methods in case of developers use the same vocabulary across source code elements and also there is enough textual source code information (i.e., a large number of covered API's methods). On average 74% and up to 76% (for swing and awt APIs, respectively) of API's methods in an identified usage pattern using combined heuristic are always uniformly co-used together.

5.2 Identified patterns from API client applications perspective (RQ2)

To answer the RQ2, NCBUPID and IML-FUP approaches are applied for identifying usage patterns from studied APIs. Then, the obtained results are compared as follows.

5.2.1 Average PUC

Table 6 shows average cohesion values for all identified usage patterns for each studied API by applying the two approaches (NCBUPID and IML-FUP). The results shown in this table reveal that both NCBUPID and IML-FUP identify patterns that have high usage cohesion values. This means that such patterns have cohesive co-usage relationships among their methods. In fact, the average cohesion values of identified patterns are around 75% for NCBUPID (namely, for both structural heuristic and a combination of structural and textual heuristics in swing and awt APIs) and 100% for IML-FUP. In spite of the results of NCBUPID are, as expected, slightly less than the results of IML-FUP, they are higher enough, taking into account the identification process of NCBUPID does not depend on client applications. Hence, the performance of NCBUPID for identifying usage patterns is comparable to IML-FUP.

Table 6 Average cohesion of identified API patterns for NCBUPID and IML-FUP

API	*NCBUPID (Str. Heu.)*	*NCBUPID (Tex. Heu.)*	*NCBUPID (Com. Hue.)*	*IML-FUP**
security	0.70	0.63	0.62	1.0
httpclient	0.70	0.63	0.51	1.0
swing	0.74	0.73	0.76	1.0
awt	0.76	0.60	0.76	1.0

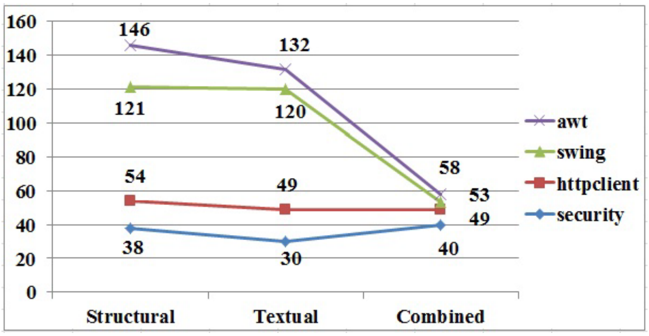
Notes: Str, Tex, Com and Hue abbreviations for Structural, Textual and Combined heuristics respectively.

IML-FUP*: Average cohesion values for core patterns identified by IML-FUP (the best average cohesion values).

5.2.2 Average number of identified patterns

Figure 10 shows the accumulative number of identified patterns using the three different heuristics. As shown in this figure, there is an order relation between numbers of identified patterns using different heuristics. For all APIs of interest, it can be noticed that structural heuristic helps to identify the largest number of usage patterns comparing with the results of applying other heuristic. Here, the number of identified patterns reached a peak of 146 patterns. In the second place, when textual heuristic is used alone, the number of identified patterns at the peak is 132 patterns. The lowest number of usage patterns is obtained when a combination of structural and textual heuristics are applied where the number of patterns at the peak is 58.

Figure 10 Average number of identified patterns using different heuristics



5.2.3 Average size of identified patterns

Figure 11 shows the accumulative number of average sizes of identified patterns using the three different heuristics. As shown in this figure, the largest usage patterns are identified when the combined heuristic is applied and size of patterns identified by applying structural and textual heuristics individually are much less than the combined heuristic. Indeed, the size of patterns identified using the combined heuristic at the peak is 98.16%. This interprets the small number of identified patterns using this heuristic.

Figure 11 Average size of identified patterns using different heuristics

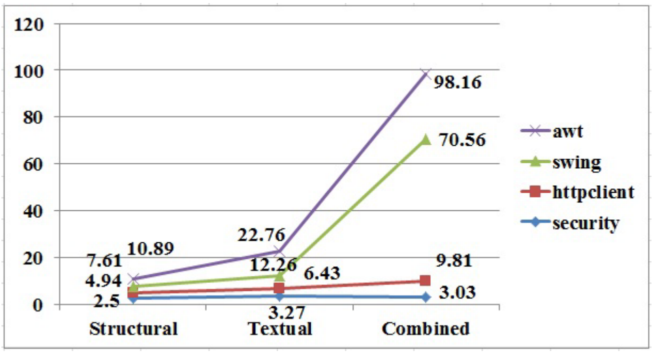


Table 7 shows the size and number of usage patterns identified from all studied APIs using both NCBUPID and IML-FUP approaches. In this table, the results of structural similarity only presented as it is better than textual and combined heuristics in terms of the number and size of usage patterns. As it is shown in this table, IML-FUP identifies more patterns than NCBUPID. Also, IML-FUP's patterns are slightly larger than NCBUPID ones. This is because IML-FUP depends on client applications to identify usage patterns. In fact, these client applications can be considered as usage scenarios of API methods. Such scenarios represent valuable information to IML-FUP for identifying API's methods that do not use to the same vocabulary, but they are frequently used together in these scenarios. The number and size of usage patterns identified using NCBUPID is comparable to ones identified using IML-FUP.

Table 7 Statistics for patterns identified from each API using NCBUPID (namely structural heuristic) and IML-FUP

API	*NCBUPID (Structural Heu.)*				*IML-FUP*			
		UP's Size				*UP's Size*		
	#UPs	*Avg.*	*Min*	*Max*	*#UPs*	*Avg.*	*Min*	*Max*
security	38	2.45	2	10	37	3.86	2	25
httpclient	16	2.44	2	5	36	2.83	2	9
swing	67	2.67	2	11	100	2.76	2	14
awt	25	3.28	2	19	60	2.61	2	6

5.3 Results summary and discussion

The findings of evaluation show that the structural heuristic and the combination of structural and textual heuristics help to identify highly cohesive API usage patterns. For all studied APIs, structural heuristic performs better than textual heuristic. However, when the developers use the same vocabulary to name API source code elements, the combination of structural and textual heuristics is the best for identifying good quality API usage patterns. This is because each heuristic represents a source of information, and in such combination the information from one source is used to filter results from another source.

The identified usage patterns using structural and combined heuristics are enough cohesive patterns to provide informative co-usage relationships between API's methods, where client applications are not available. In fact, the identified usage patterns retain informative for other clients and independent of the API usage scenarios. Consequently, the proposed approach can be used for facilitating the development tasks when new APIs are used and for enriching APIs documentation with co-usage relationships.

The proposed approach is compared with IML-FUP which is the most relevant and recent client-based work in the domain in terms of size, number and cohesion of identified usage patterns across four APIs (Salman, 2017). This work identifies usage patterns that are strongly cohesive and generalisable across different client applications. The results indicate that the proposed approach is comparable with IML-FUP in terms of the previously defined comparison criteria.

To evaluate the efficiency of NCBUPID, it is applied to four widely used APIs with 89 client applications. Additionally, the strongest points in the proposed approach as follows:

1 It follows a fully-automatic process that does not need any manual intervention during the identification process.

2 The only input required for the approach is source codes of APIs which are always available.

3 The approach is designed based on exploiting pre-proven algorithms, like breadth first search (BFS) and hierarchical clustering algorithm.

5.4 Threats to validity

As with any empirical evaluation-based work, the proposed work has internal and external threats to validity. These threats as follows:

- The identification process by using the combined heuristic is based on textual similarity between source code elements of APIs. In some situations, APIs' developers may use different vocabulary to name the source code elements which degrade the results and impact the internal validity.
- The experimental evaluation relies on a limited number of APIs and client applications, and for a better test a large number of APIs and their clients are needed. However, the APIs and their clients used in this empirical study are enough to generalise the results.
- NCBUPID is validated using only Java code (all APIs and client applications are written using Java code). However, it can be applied to any object-oriented APIs. This is because NCBUPID relies on common object-oriented elements and relationships.

6 Related work

Recently, a body of research work has been proposed to deal with APIs usage from different aspects. Robillard et al. (2013) analyse and study a large number of approaches that aim to automate the process of inference properties and knowledge from APIs. These approaches are organised into different categories based upon the goal they pursue (Robillard et al., 2013; Saied et al., 2015b): suggesting usage examples of API, suggesting API Elements, API bug detection, API usage obstacles detection and mining API usage patterns.

6.1 Suggesting usage examples of API

Usage examples are important for understanding APIs usage. Many systems were proposed in this direction. These systems are classified into two types. Firstly, IDE-based recommendation systems that suggest usage examples to developers by exploring the current context used by IDE (Perepletchikov et al., 2010; Duala-Ekoko and Robillard, 2011). As such examples are context-based examples, they cannot be used to the purpose of API documentation. Secondly, JavaDoc-based recommendation systems that can be accessed via web so they have high reachability and scalability (Holmes et al., 2006; Wang et al., 2011; Montandon et al., 2013). They aim to instrument the documentation of a given API with examples. However, they have lower precision level than IDE-based recommendation systems.

6.2 Suggesting API elements

Suggesting API elements for programmers or source code completion is a key feature in integrated development environments (IDEs). This is because it allows programmers to access API elements and free them to remember every detail. Therefore, many techniques are proposed to enhance current source code completion systems (sccs) to work with large APIs (Buse and Weimer, 2012; Bruch et al., 2009). Bruch et al. (2009) propose a system (called ICCS) that learns from source code repositories. Their work aims to filter out and evaluate the relevance of every API method suggested by an source code completion system. Nguyen et al. (2012) propose a sccs (called GrePacc). The system work by extracting features from the source code. Then, these features are exploited to find a ranked list of usage patterns that relevant to that source code.

6.3 Bugs detection

Bugs detection is an important benefit for pattern identification. For instance, suppose that API methods (*open*() and *close*()) should be called together in a given body of a function. Then, missing one of these methods is an important indicator of a bug. In the literature, there are many API bugs detection techniques (Hou and Pletcher, 2011; Nguyen et al., 2012). Li and Zhou (2005) propose PR-Miner tool. This tool is used to detect rules (or patterns) from source code repositories. Such rules can be exploited for violations detections. Monperrus et al. (2010) propose a tool (called DMMC) based on statistical information of type-usages. Each type-usage represents a list of method invocations on a variable occurring within a given method body. Such statistics are exploited to determine client methods that call the missing method.

6.4 API usage obstacles detection

Nowadays, there are many approaches that are directed to exploit developers' questions and inquiries in forums, newsgroup and Q&A website (such as stackoverflow.com), to detect usage obstacles of a given API (Li and Zhou, 2005). These approaches aim to help API developers to find out the obstacles that hinder the programming efforts and then make the required improvements. Hou and Li (2011) propose an approach to analyse discussions of 172 programmers. Then, obstacles and probable causes of API of interest are identified and described in detail. Wang and Godfrey (2013) propose to analyse developer questions of Android and iOS. Their work aims to exploit developers' posts for discovering usage scenarios of API classes that cause usage obstacles. However, they discovered a few scenarios.

6.5 API usage patterns identification for documentation and understanding purposes

The approaches that are relevant to the one proposed in this paper are those interested in API usage patterns identification for APIs documenting and understanding purposes (Nguyen et al., 2009; Michail, 1999; Michail, 2000; Uddin et al., 2012; Wang et al., 2013).

These approaches identified three categories of API usage patterns: (1) temporal, (2) sequential and (3) unordered usage patterns. The identified patterns were assessed using *consistency*, *cohesion*, *coverage* and *succinctness*. However, these approaches do not work without client programs input to the identification process. Thus, such approaches are limited to widely used APIs and cannot be applied on newly released APIs.

In the same vein, namely independence from client programs, Zhu et al. (2014) proposed an approach to automatically mining API usage examples from test code. Their proposal separated multiple test scenarios in test code and extract code examples. Then, a clustering technique is employed to assemble examples of the similar API usage for the recommendation. Like for client programs, test code is not always available for APIs which are new. Moreover, test code does not cover all functionalities of APIs that are not yet widely used.

Recently, Saied et al. (2015a) proposed an approach called *NCBUP-miner* to inferring API usage patterns using API source code, independently of the availability of API client programs. Their approach is based on a clustering algorithm called DBSCAN. The main drawback of Saied et al.'s approach is that their approach depends on an expert intervention during the inferring process to adjust values of two important parameters for hierarchical DBSCAN algorithms called *epsilon* and *epsilonStep*. The benefits of NCBUPID over their approach are as a follows. First, NCBUPID overcomes the previous mentioned limitation of NCBUPminer. Second, NCBUPID identifies usage patterns that are more highly cohesive than the ones identified by NCBUP-miner. Finally, NCBUPID helps to document the identified usage patterns by automatically generating their purpose using some heuristics based on well accepted code convention.

7 Conclusions

An approach, called *NCBUPID*, is proposed to understand and facilitate API usage through identifying unordered frequent usage patterns. The proposed approach works independently from available client applications of API of interest as these clients are not always available for new released and not widely used 490 APIs. In particular, it mainly relies on structural and textual information among

API methods to identify usage patterns using agglomerative hierarchical clustering.

An experimental evaluation using four well-known APIs is conducted. The results obtained are comparable with those approaches that depend on the availability of client applications in terms of the cohesive of identified usage patterns.

References

Alur, R., Cerný, P., Madhusudan, P. and Nam, W. (2005) 'Synthesis of interface specifications for java classes', *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Vol. 40, No. 1, pp.98–109. doi:10.1145/1047659.1040314.

Bansal, A.J. and Malhotra, R. (2016) 'Software change prediction: a literature review', International Journal of Computer Applications in Technology, Vol. 54, No. 4, pp.238–258. doi:10.1504/ijcat.2016.10001317.

Bruch, M., Monperrus, M. and Mezini, M. (2009) 'Learning from examples to improve code completion systems', *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE'09)*, ACM, New York, NY, USA, pp.213–222. doi:10.1145/1595696.1595728.

Buse, R.P.L. and Weimer, W. (2012) 'Synthesizing api usage examples', *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, IEEE Press, Piscataway, NJ, USA, pp.782–792.

Duala-Ekoko, E. and Robillard, M.P. (2011) 'Using structure-based recommendations to facilitate discoverability in apis', *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP'11)*, Springer-Verlag, Berlin, Heidelberg, pp.79–104.

Frakes, W.B. and Kang, K. (2005) 'Software reuse research: status and future', *IEEE Transaction Software Engineering*, Vol. 31, No. 7, pp.529–536. doi:10.1109/TSE.2005.85.

Haifeng, Z. and Zijie, Q. (2010) 'Hierarchical agglomerative clustering with ordering constraints', *3rd International Conference on Knowledge Discovery and Data Mining*, pp.195–199.

Holmes, R., Walker, R.J. and Murphy, G.C. (2006) 'Approximate structural context matching: an approach to recommend relevant examples', *IEEE Transaction Software Engineering*, Vol. 32, No. 12, pp.952–970. doi:10.1109/TSE.2006.117.

Hou, D. and Li, L. (2011) 'Obstacles in using frameworks and apis: an exploratory study of programmers' newsgroup discussions', *IEEE 19th International Conference on Program Comprehension (ICPC'11)*, pp.91–100. doi:10.1109/ICPC. 2011.21.

Java security api (2016) Available online at: https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html (accessed on 26 November 2016).

Java swing api (2016) Available online at: http://docs.oracle.com/javase/7/docs/api/java/security/package-summary.html (accessed on 26 November 2016).

Kayarvizhy, N., Kanmani, S. and Uthariaraj, V.R. (2016) 'Enhancing the fault prediction accuracy of ck metrics using high precision cohesion metric', International Journal of Computer Applications in Technology, Vol. 54, No. 4, pp.290–296. doi:10.1504/ijcat.2016.080493.

Kodhai, E. and Kanmani, S. (2016) 'Method-level incremental code clone detection using hybrid approach', *International Journal of Computer Applications in Technology*, Vol. 54, No. 4, pp.279–289. doi:10.1504/IJCAT.2016.10001322.

Li, Z. and Zhou, Y. (2005) 'Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code', *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Vol. 30, No. 5, pp.306–315. doi:10.1145/1095430.1081755.

Mandelin, D., Xu, L., Bodík, R. and Kimelman, D. (2005) 'Jungloid mining: helping to navigate the API jungle', *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Vol. 40, No. 6, pp.48–61. doi:10.1145/1064978.1065018.

Marcus, A. and Maletic, J.I. (2003) 'Recovering documentation-to-source-code traceability links using latent semantic indexing', *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, IEEE Computer Society, Washington, DC, USA, pp.125–135.

Michail, A. (1999) 'Data mining library reuse patterns in user-selected applications', *14th IEEE International Conference on Automated Software Engineering (ASE'99)*, pp.24–33. doi:10.1109/ASE.1999.802089.

Michail, A. (2000) 'Data mining library reuse patterns using generalized association rules', *Proceedings of the International Conference on Software Engineering*, pp.167–176. doi:10.1145/337180.337200.

Monperrus, M., Bruch, M. and Mezini, M. (2010) 'Detecting missing method calls in object-oriented software', *Proceedings of the 24th European Conference on Object-oriented Programming (ECOOP'10)*, Springer-Verlag, Berlin, Heidelberg, pp.2–25.

Montandon, J.E., Borges, H., Felix, D. and Valente, M.T. (2013) 'Documenting apis with examples: lessons learned with the api miner platform', *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE'13)*, pp.401–408. doi:10.1109/WCRE. 2013.6671315.

Nguyen, A.T., Nguyen, T.T., Nguyen, H.A., Tamrawi, A., Nguyen, H.V., Al-Kofahi, J. and Nguyen, T.N. (2012) 'Graph-based pattern oriented, context-sensitive source code completion', *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, IEEE Press, Piscataway, NJ, USA, pp.69–79.

Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M. and Nguyen, T.N. (2009) 'Graph-based mining of multiple object usage patterns', *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE'09)*, ACM, New York, NY, USA, pp.383–392. doi:10.1145/1595696.1595767.

Perepletchikov, M., Ryan, C. and Tari, Z. (2010) 'The impact of service cohesion on the analyzability of service-oriented software', *IEEE Transactions on Services Computing*, Vol. 3, No. 2, pp.89–103.

Robillard, M.P., Bodden, E., Kawrykow, D., Mezini, M. and Ratchford, T. (2013) 'Automated api property inference techniques', *IEEE Transaction Software Engineering*, Vol. 39, No. 5, pp.613–637. doi:10.1109/TSE.2012.63.

Roy, C.K., Eishita, F.Z. and Zibran, M.F. (2011) 'Useful, but usable? Factors affecting the usability of APIs', *18th Working Conference on Reverse Engineering*, pp.151–155. doi:ieeecomputersociety.org/10.1109/WCRE.2011.26.

Saied, M.A., Abdeen, H., Benomar, O. and Sahraoui, H. (2015a) 'Could we infer unordered api usage patterns only using the library source code?', *Proceedings of the IEEE 23rd International Conference on Program Comprehension (ICPC'15*), IEEE Press, Piscataway, NJ, USA, pp.71–81.

Saied, M.A., Benomar, O., Abdeen, H. and Sahraoui, H. (2015b) 'Mining multi-level API usage patterns', *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp.23–32. doi:10.1109/SANER.2015.7081812.

Salman, H.E. (2017) 'Identification multi-level frequent usage patterns from {APIs}', *Journal of Systems and Software*, Vol. 130, pp.42–56. https://doi.org/10.1016/j.jss.2017.05.039.

Thummalapenta, S. and Xie, T. (2007) 'Parseweb: a programmer assistant for reusing open source code on the web', *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ACM)*, New York, NY, USA, pp.204–213. doi:10.1145/1321631.1321663.

Uddin, G., Dagenais, B. and Robillard, M.P. (2012) 'Temporal analysis of api usage concepts', *34th International Conference on Software Engineering (ICSE'12),* pp.804–814. doi:10.1109/ICSE.2012.6227138.

Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T. and Zhang, D. (2013) 'Mining succinct and high-coverage api usage patterns from source code', *10th IEEE Working Conference on Mining Software Repositories (MSR'13)*, pp.319–328. doi:10.1109/MSR.2013.6624045.

Wang, L., Fang, L., Wang, L., Li, G., Xie, B. and Yang, F (2011) 'Api example: an effective web search based usage example recommendation system for java apis', *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*, pp.592–595. doi:10.1109/ASE.2011.6100133.

Wang, W. and Godfrey, M.W. (2013) 'Detecting api usage obstacles: a study of ios and android developer questions', *10th IEEE Working Conference on Mining Software Repositories (MSR'13)*, pp.61–64. doi:10.1109/MSR.2013.6624006.

Warintarawej, P., Huchard, M., Lafourcade, M., Laurent, A. and Pompidor, P. (2015) 'Software understanding: automatic classification of software identifiers', *Intelligent Data Analysis*, Vol. 198, No. 64, pp.761–768.

Zhong, H., Xie, T., Zhang, L., Pei, J. and Mei, H. (2009) 'Mapo: mining and recommending API usage patterns', *Proceedings of the 23rd European Conference on (ECOOP'09) – Object-Oriented Programming*, Genoa, Italy, Springer-Verlag, Berlin, Heidelberg, pp.318–343. doi:10.1007/978-3-642-03013-0_15.

Zhu, Z., Zou, Y., Xie, B., Jin, Y., Lin, Z. and Zhang, L. (2014) 'Mining api usage examples from test code', *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*, IEEE Computer Society, Washington, DC, USA, pp.301–310. doi:10.1109/ICSME.2014.52.

Notes

1 http://hc.apache.org/httpclient-3.x/

2 http://docs.oracle.com/javase/7/docs/api/java/security/package-summary.html

3 http://docs.oracle.com/javase/7/docs/technotes/guides/swing/

4 http://docs.oracle.com/javase/7/docs/api/

5 http://docs.oracle.com/javase/7/docs/api/javax/swing/GroupLayout.html

6 https://docs.oracle.com/javase/7/docs/api/java/security/KeyStore.html

7 https://sourceforge.net/, https://github.com/, https://code.google.com/archive/search and https://gitorious.org/