Hindawi Publishing Corporation
Scientific Programming
Volume 2016, Article ID 5475964, 11 pages
http://dx.doi.org/10.1155/2016/5475964

*Research Article*

# Mining Software Repositories for Automatic Interface Recommendation

**Xiaobing Sun,[1,2] Bin Li,[1,2] Yucong Duan,[3] Wei Shi,[1] and Xiangyue Liu[1]**

[1]*School of Information Engineering, Yangzhou University, Yangzhou 225127, China*
[2]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China*
[3]*Hainan University, Haikou 570228, China*

Correspondence should be addressed to Xiaobing Sun; sundomore@163.com

Received 30 January 2016; Revised 3 May 2016; Accepted 18 May 2016

Academic Editor: Laurence T. Yang



There are a large number of open source projects in software repositories for developers to reuse. During software development and maintenance, developers can leverage good interfaces in these open source projects and establish the framework of the new project quickly when reusing interfaces in these open source projects. However, if developers want to reuse them, they need to read a lot of code files and learn which interfaces can be reused. To help developers better take advantage of the available interfaces used in software repositories, we previously proposed an approach to automatically recommend interfaces by mining existing open source projects in the software repositories. We mainly used the LDA (Latent Dirichlet Allocation) topic model to construct the Feature-Interface Graph for each software project and recommended the interfaces based on the Feature-Interface Graph. In this paper, we improve our previous approach by clustering the recommending interfaces on the Feature-Interface Graph, which can recommend more accurate interfaces for developers to reuse. We evaluate the effectiveness of the improved approach and the results show that the improved approach can be more efficient to recommend more accurate interfaces for reuse over our previous work.

## 1. Introduction

Interfaces are an integral part of software projects, which is also an important means for software design. During software development, most of the software projects use interfaces to construct the frame of a project [1–3]. Program interface is an effective way to construct the framework of a new project. Reusing interfaces in software projects can save development time and reduce development cost. Moreover, most of these open source projects are of high quality [4–8]. If we can reuse the interfaces in these open software projects to help develop a new project, it can effectively allow developers to write less code and improve the efficiency of their project development [9–12].

Today, many open source software repositories are available to developers such as SourceForge (https://sourceforge.net/), Git (https://git-scm.com/), and OSChina (http://www.oschina.net/). These software repositories provide an amount of useful information to developers such as coding style, document, and software interfaces. Hence, mining existing software repositories can help developers to identify some interfaces for reuse. One simple and direct method is to check these code files one by one and then extract the files useful for developers. However, it is difficult, costly, and labor-intensive for developers to implement interfaces all by themselves for a new project.

The main problem is how we can deal with these useful information easily. A number of techniques have been proposed to analyze the software repositories for reuse in software development. Chan et al. proposed a code search approach, which returns a graph of API methods [13]. They used a greedy subgraph search algorithm to identify a connected subgraph containing nodes with high textual similarity to the query phrases. Thung et al. proposed a library recommendation approach based on a set of libraries that a project currently uses, which recommends relevant libraries [14, 15]. They proposed a hybrid approach that combines association rule mining and collaborative filtering to recommend libraries for reuse. In addition, they also proposed an API recommendation approach [15]. The APIs are recommended

based on the measure of the relevance by looking into the similarity between description of feature request and description of APIs. However, the search range of these techniques is too large and the process of matching is complex. Moreover, few of them focus on interface recommendation.

Considering the above issues, an automated approach is needed to recommend interfaces to developers more effectively. In our previous work, we proposed an approach to effectively mine the software repositories for automatic interface recommendation [16]. The approach used the LDA (Latent Dirichlet Allocation) topic model [17] to construct the Feature-Interface Graph (FIG) for each project to recommend interfaces based on their usage in each project. However, the number of recommended interfaces is too large, and we only consider the individual open software project for recommendation. But in practice, different projects have different considerations; that is, some interfaces are more important in some projects while some others are less in other projects. In addition, the time to construct the Feature-Interface Graph (FIG) for each project is too large, and developers need to wait a long time to reuse these recommended interfaces. In this paper, we improve our previous work considering these two aspects in the following way. On the one hand, we use the LDA topic model to extract the topics of open source projects in the open source software repositories and identify the relevant open source projects. In this way, we do not need to construct the FIGs for each project in the software repositories, which is costly. On the other hand, we cluster the interfaces of these relevant projects based on the FIG and recommend the interfaces on the clustering results. In this way, the interfaces are recommended based on their general usage in all of the relevant software projects in software repositories. In this paper, we make the following contributions:

(i) We improve our previous work and propose a more effective approach for automatic interface recommendation by mining software repositories.

(ii) We conduct an empirical study to show the performance improvement of our approach, and the empirical results show that the performance of our approach is greatly improved over our previous work.

(iii) We conduct an empirical study to show the accuracy improvement of our approach, and the empirical results show that the accuracy of the recommended interfaces is also greatly improved over our previous work.

The rest of the paper is organized as follows. In Section 2, we introduce the background related to the LDA topic model. We present our approach in Section 3. In Section 4, we conduct an empirical study to evaluate our approach. In Section 5, we discuss the related work. Finally, Section 6 concludes our approach and discusses some future work.

## 2. Background

Topic models were originated from the field of information retrieval (IR) to index, search, and cluster a large amount of unstructured and unlabeled documents. A topic is a collection of terms that cooccur frequently in the documents of the corpus. One of the mostly used topic models in software engineering community is Latent Dirichlet Allocation (LDA) [17]. It has been successfully and widely applied to analyze the software engineering data to support various software engineering activities [18–22].

LDA is a probabilistic generative topic model for collections of discrete data such as text corpora [17]. It has been applied to information retrieval to automatically organize and provide structure to a text corpus [23]. In LDA, there is a set of topics to describe the entire corpus. Each document can contain more than one of these topics, and each term in the entire repository can be contained in more than one of these topics. Hence, LDA is able to discover a set of ideas or themes that well describe the entire corpus. In this paper, we extract the topics from the software by using the LDA topic model. These extracted topics help us understand the features of the target software.

The basic main idea of LDA is that documents are represented as random mixture over latent topics, where each topic is characterized by a distribution over words [17]. Specifically, the LDA model consists of the following building blocks:

(1) A word is the basic unit of discrete data, defined to be an item from a software vocabulary $V = \{w_1, w_2, \ldots, w_v\}$, such as an identifier or a word from a comment.

(2) A document is a sequence of words denoted by $d = \{w_1, w_2, \ldots, w_n\}$, where $w_i$ is the $i$th word in the sequence.

(3) A corpus is a collection of documents denoted by $D = \{d_1, d_2, \ldots, d_m\}$.

Given $m$ documents containing $k$ topics expressed over $v$ unique words, the distribution of $i$th topic $t_i$ over $v$ words and the distribution of $j$th document over $k$ topics can be represented.

By applying the LDA model, it outputs the distribution values of the topics of the software projects. We use the distribution values to show the accuracy of the topic. More details of LDA can refer to the work of Blei et al. [17].

## 3. Our Approach

In this section, we present our approach about how to automatically recommend interfaces. To develop a new software, one of the first tasks is to specify the function modules based on requirement analysis. During this process, interface definition is an important task. After specifying the function modules and providing the features about the new program, some suitable interfaces should be defined. If some of the interfaces can be recommended for developers to reuse, it will improve the efficiency of software development and increase the opportunity of code reuse.

The process of our approach is shown in Figure 1. The input of our approach includes the software repositories and the feature request of a new project. We first use the LDA topic model to extract the software projects from software

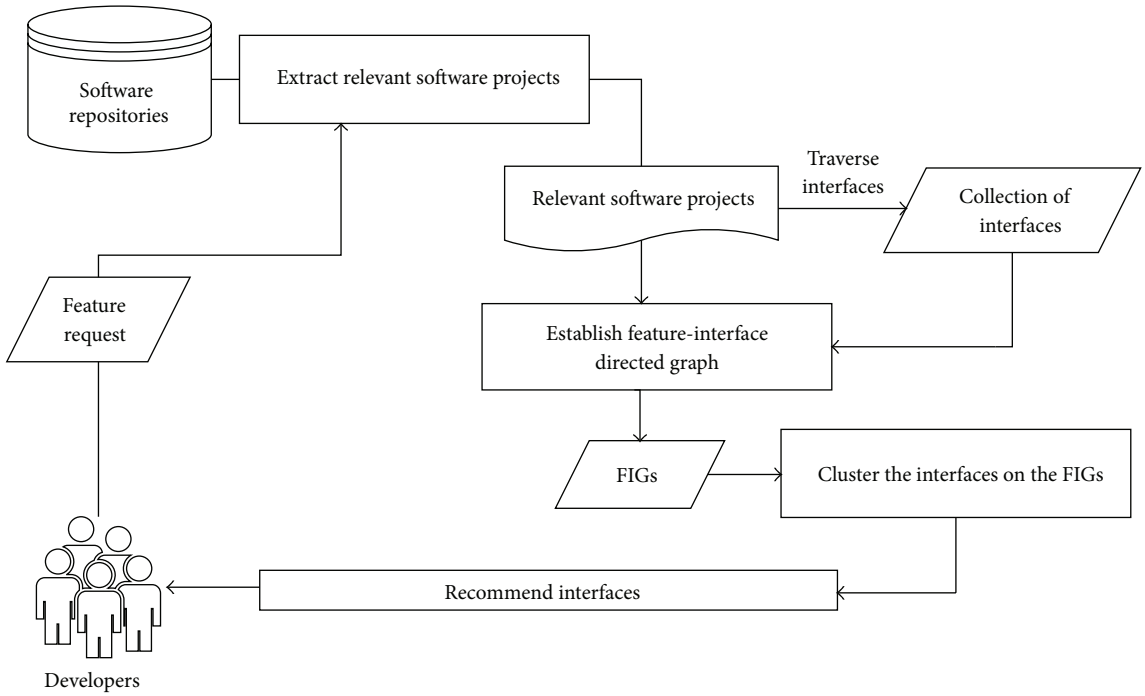



Figure 1: Process of our approach.

Table 1: The topics and the allocation values of topics extracted by LDA from JEdit project.

| Topic | Allocation | Word |
|---|---|---|
| Topic 0 | 0.2722 | textarea, jedit, gnu, general, software, version, line, copy, color, foundation |
| Topic 1 | 0.1836 | jedit, entry, code, path, view, url, tag, gjt, jar, title |
| Topic 2 | 0.2748 | type, undo, scan, scanpos, stream, read, input, encoding, dimision, font |
| Topic 3 | 0.2401 | jedit, view, box, options, actionhandler, cons, selected, borderlayout, attribute, entity |
| Topic 4 | 0.2088 | start, end, line, selection, offset, match, input, caret, search, strtline |
| Topic 5 | 0.1740 | keyevent, vk, key, jedit, length, text, shortcut, installer, historymodel, swing |
| Topic 6 | 0.1700 | path, vfs, browser, file, jedit, view, directory, session, menu, component |
| Topic 7 | 0.2342 | offset, line, length, file, start, seg, header, jedit, number, rule |
| Topic 8 | 0.2379 | interpreter, callstack, file, type, primitive, pat, lgpl, version, notice, item |
| Topic 9 | 0.6456 | jj, active4, 3r, jjtn, xsp, active4, active2, jjtree, active1, active0 |

repositories that are relevant to the feature request provided by developers. Then, we traverse the interfaces of these relevant software projects and construct the Feature-Interface Graphs (FIGs) for each of the relevant project. Finally, based on the interfaces on the FIGs, we cluster the interfaces and recommend the candidate interfaces based on the number of their invocations in software repositories.

*3.1. Identify Relevant Software Projects.* First, we use the topic model (LDA) to extract the topics for each project in software repositories. After extracting the topics, we get some words to describe each open source project. Before using the LDA, we should preprocess these software projects to reduce noise and improve the data quality for use. There are several typical natural language processing (NLP) preprocessing operations for the unstructured source code. We tokenize each word based on common naming practices, such as camel case (*oneTwo*) and underscores (*one_two*), remove common English language stop words (*the*), and strip away syntax and programming language keywords (*public*). More details of the preprocessing operations can refer to our previous work [24]. After preprocessing, we can use the LDA to extract the topics for each project. We see each project as one document collection. For each project, we get $K$ topics to describe the features and $K$ topic allocation values, we denote these topics as a collection named $F_i$. Table 1 shows an example of the topics extracted from JEdit (http://www.jedit.org/), a famous text editor open source software. There are ten topics extracted from this project. The first column is the topics, the second the allocation of each topic which describes the significance of this topic, and the last column the words describing the topic.

At the beginning of the development process, developers may provide a functional description of their request. The description words of the new software are marked as

a collection. For the feature request of a project, it is a short textual description. So tokenization of the feature request is conducted. During the process of tokenization, the feature request text is turned into tokens. Then, we also get a set of words to represent the feature request; that is, Feature = $\{w_1, w_2, \ldots, w_k\}$. Then, we turn to match the words in the collection with the topics of each software project; that is, $\text{Software}_i = \{w_1, w_2, \ldots, w_j\}$. Specifically, for each project in the software repository, we compute the similarity value ($\text{Sim}_i$) by measuring how many words of the feature request are also present in the LDA representation of a software ($\text{Software}_i$), as shown as the following formula:

$$\text{Sim}_i = \frac{\left|\text{Feature} \cap \text{Software}_i\right|}{\text{Software}_i}. \quad (1)$$

The greater value the similarity ($\text{Sim}_i$) is, the higher the similarity between feature request of the new project and the target project in software repositories is. If the value of the counter satisfies a threshold value, we claim that the target project ($\text{Software}_i$) is relevant to the new project at hand.

*3.2. Feature-Interface Graph Construction.* In this paper, Feature-Interface Graph (FIG) is an important representation for interface recommendation. In this subsection, we discuss the definition of FIG and the process of how to construct FIG.

FIG is formally defined as follows.

*Definition 1* (Feature-Interface Graph). The Feature-Interface Graph (FIG) is a graph; FIG = $\langle N, E, W \rangle$. $N = N_1 \cup N_2$, where $N_1$ is the topic nodes and $N_2$ is the interface nodes corresponding to the topic nodes. $E$ is the relation between topic nodes and interface nodes, which is identified by the LDA. $W$ is the weight on the edge, which represents the number of interfaces been invoked in the software.

So on the FIG, there are two types of nodes, that is, topic nodes and interface nodes. Topic nodes are used to store the topics of the software. Interface nodes are the nodes of interfaces used in the software. We distinguish topic nodes and interface nodes because when developers need interfaces to construct the frame of a new project, they need to first provide the feature request about this new project. Then, we can retrieve the interface nodes through matching the topic nodes with the description of the new project.

Then, we traverse each project file and search interfaces used in this project. We store the interfaces in the form of *Project Name-Package Name-File Name-Interface Name*. For example, we find an interface named *BufferChangeListener*. We record its information and store this interface in the form of *jedit-org.gjt.sp.jedit.buffer-BufferChangeListener*. To show the importance of the interfaces, we count the number of interfaces being invoked. Table 2 shows the interfaces in the JEdit and the number of these interfaces being invoked.

After extracting the topics from the relevant projects and traversing the interfaces, their FIGs can be constructed. We mark the collection of topics as the topic nodes, the collection of interfaces as interface nodes, and the number of interfaces being invoked as the weight of edges. Figure 2 shows an example of the FIG for the *JEdit* project. In Figure 2, there are many relevant open source projects in software repositories. For each project, it has its own FIG and has ten topics to describe its features and functions. In our approach, we choose some of these topics to be considered as the most suitable description of *JEdit*. For each project, we also search the interfaces used in corresponding projects and calculate the number of them being invoked. In Figure 2, the weight of edges represents the number of interfaces being invoked.

*3.3. Interface Clustering and Recommendation.* After constructing the FIGs for each of the relevant projects, we can obtain the information about the topics of the software and their interfaces and the number of each interface being invoked in the software. In our previous work, the interfaces are recommended based on the number of their being invoked for each project. However, different projects have their individual characteristics and the usage of interfaces is different. If we only consider the interface usage in individual project, the recommending results may be inaccurate. So in this paper, the interfaces are recommended based on the whole perspective of interface usage in all of the relevant software projects.

We cluster the interfaces also based on the LDA topic model. Each interface on the FIGs is considered as a file for LDA input. The process of interface clustering includes the following steps:

(1) Natural language processing (NLP) techniques are used to preprocess the interfaces to reduce the noise in the interface data.

(2) LDA is applied to generate the topics.

(3) Clusters are generated, and interfaces having similar topics should be allocated in a cluster.

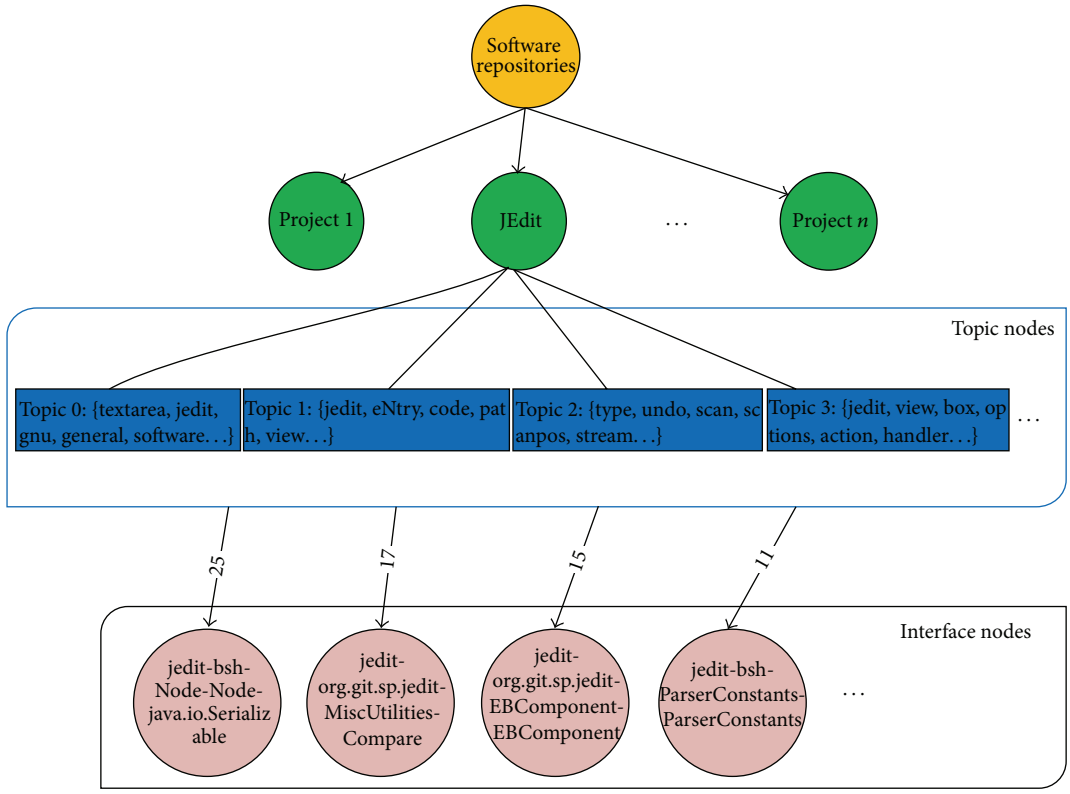
Based on the interface clustering, we can compute the total count of the interfaces being invoked in each cluster. The greater value the count is, the higher the interface in that cluster relevant to a topic is concerned. When this step is finished, the interfaces are ranked and recommended based on the following formula:

$$\text{Rank}\left(\text{Inter}_i\right) = \text{Invoke}\left(\text{Inter}_i\right) \times \frac{\text{Cluster}\left(\text{Inter}_i\right)}{\text{Max}\left(\text{Clustering}\right)}. \quad (2)$$

In (2), $\text{Invoke}(\text{Inter}_i)$ represents the number of Interface $i$ being invoked in a project; $\text{Cluster}(\text{Inter}_i)$ represents the size of the cluster Interface $i$ belonging to. LDA generates $K$ topics. For each topic, it is a cluster, which is composed of some interfaces related to this topic. So when using LDA, we need to set the number of generated topics, $K$. In addition, in Formula (2), Max(Clustering) represents the largest size of the cluster in the generated clusters. Based on this metric, the top $K$ interfaces can be recommended. According to the directed edges they points to, the interfaces can be automatically determined. Then, we go back to the corresponding project to find the code snippet of the interfaces and recommend the source code to the developers.

Table 2: The interfaces and the number of their being invoked in the JEdit project.

| Number | Interface | Invoked account |
|---|---|---|
| 1 | jedit-org.gjt.sp.jedit.browser-BrowserListener-BrowserListener | 0 |
| 2 | jedit-org.gjt.sp.jedit.buffer-BufferChangeListener-BufferChangeListener | 1 |
| 3 | jedit-org.git.sp.jedit.buffer-BufferListener-BufferListener | 3 |
| 4 | jedit-org.git.sp.jedit.gui-DefaultFocusComponent-DefaultFocusComponent | 2 |
| 5 | jedit-org.git.sp.jedit.gui-DockableWindowContainer-DockableWindowContainer | 2 |
| 6 | jedit-org.git.sp.jedit.gui-MutableListModel-MutableListModel | 2 |
| 7 | jedit-org.git.sp.jedit.help-HelpHistoryModelListener-HelpHistoryModelListener | 1 |
| 8 | jedit-org.git.sp.jedit.help-HelpViewerInterface-HelpViewerInterface | 1 |
| 9 | jedit-org.gjt.sp.jedit.indent-IndentAction-IndentAction | 5 |
| 10 | jedit-org.gjt.sp.jedit.indent-IndentRule-IndentRule | 2 |
| 11 | jedit-org.gjt.sp.jedit.menu-DynamicMenuProvider-DynamicMenuProvider | 7 |
| 12 | jedit-org.gjt.sp.jedit.search-HyperSearchNode-HyperSearchNode | 2 |
| 13 | jedit-org.gjt.sp.jedit.search-HyperSearchResults-ResultVisitor | 0 |
| 14 | jedit-org.gjt.sp.jedit.search-HyperSearchTreeNodeCallback-HyperSearchTreeNodeCallback | 2 |
| 15 | jedit-org.gjt.sp.jedit.search-SearchFileSet-SearchFileSet | 2 |
| 16 | jedit-org.gjt.sp.jedit.syntax-TokenHandler-TokenHandler | 2 |
| 17 | jedit-org.gjt.sp.jedit.textarea-ScrollListener-ScrollListener | 2 |
| 18 | jedit-org.gjt.sp.jedit.textarea-ScrollListener-java.util.EventListener | 1 |
| 19 | jedit-org.gjt.sp.jedit.textarea-StatusListener-StatusListener | 2 |
| 20 | jedit-org.gjt.sp.jedit.textarea-StructureMatcher-StructureMatcher | 0 |
| 21 | jedit-org.gjt.sp.jedit-EBComponent-EBComponent | 15 |
| 22 | jedit-org.gjt.sp.jedit-MiscUtilities-Compare | 17 |
| 23 | jedit-org.gjt.sp.jedit-OptionPane-OptionPane | 1 |
| 24 | jedit-org.gjt.sp.jedit-Registers-Register | 2 |
| 25 | jedit-org.gjt.sp.util-ProgressObserver-ProgressObserver | 3 |
| 26 | jedit-org.gjt.sp.util-WorkThreadProgressListener-WorkThreadProgressListener | 0 |
| 27 | jedit-org.objectweb.asm-ClassVisitor-ClassVisitor | 1 |
| 28 | jedit-org.objectweb.asm-CodeVisitor-CodeVisitor | 1 |
| 29 | jedit-org.objectweb.asm-Constants-Constants | 1 |
| 30 | jdeit-bsh-BSHBlock-NodeFilter | 1 |
| 31 | jedit-bsh-BshClassManager-Listener | 2 |
| 32 | jedit-bsh-BshIterator-BshIterator | 2 |
| 33 | jedit-bsh-ConsoleInterface-ConsoleInterface | 1 |
| 34 | jedit-bsh-NameSource-NameSource | 0 |
| 35 | jedit-bsh-Node-Node | 1 |
| 36 | jedit-bsh-Node-Node-java.io.Serializable | 25 |
| 37 | jedit-bsh-ParserConstants-ParserConstants | 11 |
| 38 | jedit-bsh-ParserTreeConstants-ParserTreeConstants | 1 |
| 39 | jedit-com.microstar.xml-XmlHandler-XmlHandler | 1 |
| 40 | jedit-gnu.regexp-CharIndexed-CharIndexed | 6 |
| 41 | jedit-installer-BZip2Constants-BZip2Constants | 2 |
| 42 | jedit-installer-Progress-Progress | 2 |
| 43 | jedit-installer-TarInputStream-EntryFactory | 1 |
| 44 | jedit-java.swing.border-Border | 2 |
| 45 | jedit-java.swing.text.Postion | 1 |
| 46 | jedit-javax.swing.ListModel | 1 |
| 47 | jedit-java.lang.reflect.InvocationHandler | 1 |
| 48 | jedit-java.util.Enumeration | 1 |
| 49 | jedit-java.util.EventListener-EventListener | 0 |




Green node: name of project
Blue rectangle: topics of JEdit extracted by topic model
Pink node: interfaces in JEdit
Weight of edges: number of interfaces being invoked

Figure 2: The Feature-Interface Graph of JEdit in the software repositories.

## 4. Empirical Study

In our previous work, we have evaluated the effectiveness of our approach [16]. While in this paper, we improve our previous work aiming at decreasing the time cost and improving the recommendation accuracy. So we define the following research questions to compare with our previous work.

*RQ1.* Can our approach improve the efficiency of recommending interfaces over our previous work?

*RQ2.* Can our approach improve the accuracy of the recommended interfaces over our previous work?

*4.1. Setup.* To evaluate the effectiveness of our approach, we conduct an empirical study on a famous open source software repository, that is, SourceForge (https://sourceforge.net/), to recommend interfaces.

In our approach, we need to use LDA to generate the topics of the projects and interfaces (to save time, we only select 200 active open source projects in the SourceForge repositories). For LDA computation, we used *MALLET* (http://mallet.cs.umass.edu/), which is a highly scalable Java implementation of the *Gibbs* sampling algorithm. We ran for 10,000 sampling iterations, the first 1000 of which were used for parameter optimization. During this process, we need to set the number of topics ($K$) for LDA analysis. In our previous work, we have conducted some empirical studies to show which number of topics is good for interface recommendation, and the results show that four number of topics is a relatively good choice. So in this study, we also set the number of topics $K = 4$.

In addition, to conduct our study, we invited 20 people to participate in our evaluation. These 20 participants are from all walks of life, such as developers and students. Half of them are from school with 2-3 years of development experience and the other half are from industry with 3-4 years of development experience. In our study, the task for them is to write a text-editing software as in our previous work because all of them used text-editing software frequently and they have the experience of developing this kind of software. These 20 participants are divided into two groups, and they used the previous approach [16] and the new approach in this paper to recommend interfaces, respectively. They performed the study under a similar circumstance.

For *RQ*1, we record the time of the interface recommendation approach in this paper and that in our previous work, respectively.

Table 3: The average time (minutes) and gain (over previous approach) to recommend the interfaces.

| Previous approach | New approach | Gain |
|---|---|---|
| 55.1 | 26.5 | 51.9% |

For *RQ*2, we ask the participants to provide the feature request for interface recommendation. Then, they evaluate the usefulness of these interfaces to new projects. They evaluate this by the *support* of the interfaces recommended for them, which represents the usefulness of these interfaces that can be used in their development process. The highest score is 4 and the lowest score is 0. The higher the score is, the more useful the interface is. For each participant, top twenty interfaces will be recommended for them. In addition, after their rating on the recommended interfaces, we asked them to discuss on the recommended results and get a consensus on which interfaces would be finally used to develop the new project. This final results on the potentially-be-used interfaces will be used as an authoritative result to measure our previous approach and new approach. To quantitatively compare these two approaches, we used precision and recall, two widely used information retrieval and classification metrics [25], to validate the accuracy of different approaches. Precision measures the fraction of interfaces identified by an interface recommendation approach that are truly relevant (based on the authoritative results), while recall measures the fraction of relevant results (i.e., interfaces that appear in the authoritative results).

*4.2. Results.* In this subsection, we analyze and discuss the results collected from our studies.

*4.2.1. RQ1.* In our previous work, Feature-Interface Graph (FIG) for each project in software repositories needs to be generated. While in this paper, we have a step to extract the relevant projects from software repositories and then generate FIGs for these relevant projects. However, during this process, we need to run the LDA topic model to extract the relevant software projects. Moreover, in this paper, we also need to use the LDA topic model to cluster the interfaces for interface generation. In practical software development, efficiency is an important factor for interface reuse. So the first research question is to evaluate whether the approach in this paper can improve the efficiency of our previous work.

Figure 3 shows the time of these twenty participants to get the interfaces recommended by the previous approach and new approach. We notice that the new approach always needs less time to recommend the interfaces. Table 3 shows the average time for these two approaches to generate the interfaces and the gain of new approach over previous approach. From the results, we see that, on average, the time for the new approach to generate the interfaces is 26.5 minutes and 55.1 minutes for previous approach. Moreover, the gain of the new approach over previous approach is bigger than 50%, which is a relatively big scale. Hence, the results indicate that the new approach can save much time to recommend the interfaces, which is more efficient for developers to use. So based on the time results, we can conclude that our approach improves the efficiency over the previous approach.

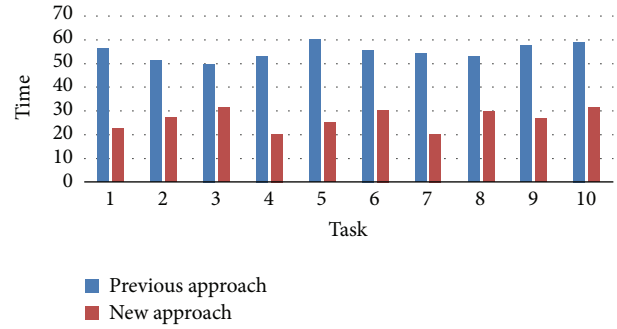



Figure 3: The time to recommend the interfaces for each participant (minutes).

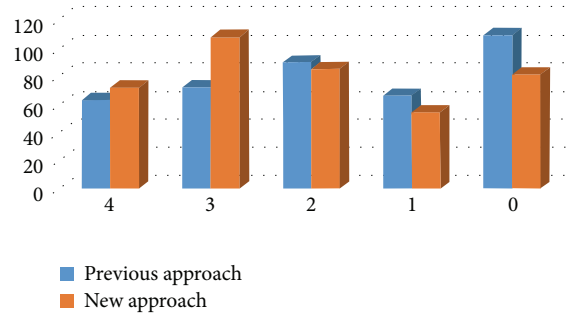



Figure 4: The support results assessed by participants (4 represents more useful, 3 useful, 2 undetermined, 1 useless, and so on).

*4.2.2. RQ2.* In the new approach, we use a different metric to rank the recommended interfaces. For previous approach, we consider the usage of interfaces in each individual project, while in the new approach, we consider the usage of interfaces from a general view on similar interfaces of the relevant projects. In our previous work, we have shown the accuracy of the recommended interfaces [16], while in the new approach, we use a different metric to recommend the interfaces. So the second research question aims to see whether the recommended interfaces of the new approach are more accurate than the previous approach.

In our study, we select the top twenty interfaces recommended by each approach for participants to assess. So for each participant, he/she needed to evaluate whether each of the twenty interfaces is useful for reuse. So there are in all 400 assessment results. Figure 4 shows the evaluation results of all the participants. From the results, we see that the number of interfaces recommended by the new approach assessed with the highest score (which indicates that the interfaces are more useful) is bigger than that of the previous approach. In addition, the number of interfaces recommended by the new approach assessed with the score (3) is also bigger than that of the previous approach. On the contrary, for interfaces with other scores, that is, 2, 1, and 0, the number of previous approach is bigger than the new approach. In addition, we use the precision and recall to quantitatively compare

Table 4: The precision and recall of these two approaches to recommend the interfaces.

| Approach | Precision | Recall |
|---|---|---|
| Previous approach | 21.2% | 44.8% |
| New approach | 27.5% | 62.4% |
| Gain | 29.7% | 39.3% |

the previous approach and new approach. The results are shown in Table 4, which show that both the precision and recall of the new approach are improved. Specifically, the gain approach of the new approach over previous approach is 29.7%, and the gain of recall is 39.3%. So from the results, we notice that the new approach can recommend more useful interfaces for participants to reuse. Hence, we can conclude that the new approach can recommend more accurate interfaces for reuse over the previous approach.

*4.3. Threats to Validity.* In our study, we have demonstrated the effectiveness of our new approach over the previous approach. But there are still some problems in our evaluation, which threaten the effectiveness of our study.

In our study, the first threat to validity comes from the participants and the project used in the evaluation. The participants in our study engaged in a similar job, but they write their own feature request for interface recommendation. Hence, they may hold different views about text-editing software like *JEdit*. Some of them do not have enough programming experience and do not understand the function of interfaces. In our evaluation, we list the interfaces and functional description of them. Participants need to read these description first and then score for the importance of these interfaces based on their own understanding. We address this threat by using popular and simple words to describe the function of the interfaces and illustrate these descriptions by giving examples. Moreover, these participants performed the study under a similar circumstance, but in practice, we cannot control that they can be fully devoted to the experiment during the empirical study.

In addition, we only choose a text-editing software for participants to conduct our study. We validate the effectiveness of our approach based on the commonly used text-editing software, which is easy and widely used for conducting studies. However, the results cannot be generalized to other projects. Hence, more studies are needed to evaluate our approach on different projects with different characteristics.

The other threat to our validity includes the repositories we used and a possible mismatching between the features provide by the developers and the topics extracted from the open source projects. In addition, the number of topics is set as 4 based on our previous work [16]. But a different value for the number of topics may generate different results.

## 5. Related Work

A number of approaches have been studied to alleviate the difficulty of code reuse, and some code/software recommendation techniques were proposed, which include the recommendations of example code [26–29], libraries [14] and APIs [30, 31], and software applications [32].

A lot of current related work in this area focused on code or code example recommendation [33–35]. Bruch et al. proposed a code completion system that recommends method calls by looking for code snippets in existing code repositories [36]. They developed this system based on the frequency of method call usage. Proksch et al. also focused on code completion and used Bayesian networks for more intelligent code completion [37]. Lozano et al. proposed a source code recommendation tool that aids developers in software maintenance. They used a genetics-inspired metaphor to analyze source code entities related to the current working context and provided its developer with a number of recommended properties such as naming conventions, used types, and invoked messages [33]. Murakami and Masuhara proposed a method that mechanically evaluates usefulness for their recommendation system called *Selene*. They adjusted several search and user-interface parameters in *Selene* for better recommendation [35]. Then, they also proposed using the user's editing activity to identify the source code relevant to the current method/class. They used a modified degree-of-interest model and incorporated the model in repository-based code recommendation system [28]. McMillan et al. proposed an approach for rapid prototyping that facilitates software reuse by mining feature descriptions and source code from open source repositories. They recommended features and associated source code modules that are relevant to the software product under development [34]. Kim et al. proposed a code example recommendation system that combines the strength of browsing documents and searching for code examples and returns high-quality code example summaries mined from the Web [27]. Ghafari et al. proposed an approach for code recommendation in which code examples are automatically obtained by mining and manipulating unit tests [26]. Zagalsky et al. developed a code search and recommendation tool which brings together social media and code recommendation systems. The tool enables crowd-sourced software development by utilizing both textual and social information [29]. Zhang et al. proposed an approach that recommends interface parameters for code completion. They used an abstract usage instance representation for each API usage example and built a parameter usage database. Then, they queried the database for abstract usage instances in similar contexts and generated parameter candidates by concretizing the instances adaptively [31]. All the above work focus on fine-code level recommendation, which can be used in code completion. In this paper, we recommend higher-level code, that is, interfaces, for developers' reuse, which is different from the above work.

There are also some work that are similar to us, which can recommend libraries or interfaces. Thung et al. proposed two techniques that recommend libraries and APIs [14, 15]. In [14], they proposed a technique that automatically recommended libraries for a particular project. Their input is a set of libraries that the project has used, and the output is the other libraries that are potentially useful for it. Our approach differs in several respects. The input of our approach is the feature of the new project. Then, our approach returns

the potentially useful APIs to the developers based on the matching results between features of the new project and the topics of existing projects. In addition, they also proposed a technique that automatically recommended APIs based on a feature request [15]. They take as input a new textual description and recommended API methods from a set of libraries. They compared the textual description of feature request with the description of the APIs in the API document. Then, they computed the final similarity score between two feature requests and recommended the best suitable APIs. Our work differs from theirs. That is, we studied the relationship between the number of APIs being invoked and the importance of them. Moreover, our approach only needed feature request as the input. Chan et al. proposed a recommendation of API methods through giving textual phrase [13]. Their approach returned a connected APIs undirected graph. Based on this graph and the input query, they mined a subgraph with a particular objective function. In our previous approach [16], we built a graph different from theirs. We recommend the interfaces based on the topic model by matching the feature request and the topics of existing projects. In this paper, we improve our previous work considering further improving the efficiency and accuracy.

There are also a few of work that focused on higher-level application recommendation. McMillan et al. proposed an approach for automatically detecting closely related applications to help users detect similar applications for a given Java application. They used an algorithm that computes a similarity index between Java applications using the notion of semantic layers that correspond to packages and class hierarchies [32]. In addition, they proposed to use API calls from third-party libraries for automatic categorization of software applications that use these API calls [8]. It enables different categorization algorithms to be applied to repositories that contain both source code and bytecode of applications. In this paper, we focused on interface recommendation, and recommendation of applications will become our future concern.

## 6. Conclusion

In this paper, we improve our previous work [16] to recommend interfaces by using LDA topic model to establish the FIG. We mainly consider two aspects to improve the previous interface recommendation technique. One is that we only identify the relevant open source projects to construct the Feature-Interface Graph to improve the efficiency of the recommendation process. The other is that the interfaces are recommended based on a general usage of the recommended interfaces in all of the relevant software projects in software repositories to improve the recommendation accuracy. The empirical results show that the improved approach in this paper can be more efficient to recommend more accurate interfaces for reuse over the previous work.

In our interface recommendation approach, we still have some issues that remain unsolved. In future, we plan to conduct more studies with some tools that are used to mine API usage and software search such as MAOP [38, 39], Apatite [40], and code quality [41]. We also plan to conduct empirical studies with a large number of other open source projects to show the generality of our approach. Finally, we would like to focus on higher-level recommendation, such as service or application recommendation [42], which considers the recommended interfaces in this paper.

## Competing Interests

The authors declare that they have no competing interests.

## Acknowledgments

This work is supported partially by Natural Science Foundation of China under Grants no. 61402396, no. 61472344, and no. 61472343, partially by the Natural Science Foundation of the Jiangsu Higher Education Institutions of China under Grant no. 13KJB520027, partially by the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University under Grant no. KFKT2016B21, partially by the Jiangsu Qin Lan Project, partially by the China Postdoctoral Science Foundation under Grant no. 2015M571489, and partially by the Nantong Application Research Plan under Grant no. BK2014056.

## References

[1] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, pp. 111–120, Honolulu, Hawaii, USA, May 2011.

[2] C. McMillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu, "Portfolio: searching for relevant functions and their usages in millions of lines of code," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 4, article 37, 2013.

[3] E. Moritz, M. L. Vasquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers, "ExPort: detecting and visualizing API usages in large source code repositories," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE '13)*, pp. 646–651, Silicon Valley, Calif, USA, November 2013.

[4] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: an infrastructure for large-scale collection and analysis of open-source code," *Science of Computer Programming*, vol. 79, pp. 241–259, 2014.

[5] M. Grechanik, C. McMillan, L. DeFerrari et al., "An empirical investigation into a large-scale Java open source code repository," in *Proceedings of the 4th International Symposium on Empirical Software Engineering and Measurement (ESEM '10)*, Bolzano, Italy, September 2010.

[6] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: a source code search engine for finding highly relevant applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1069–1087, 2012.

[7] E. Moritz, M. Linares-Vasquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers, "ExPort: detecting and visualizing API usages in large source code repositories," in *Proceedings of the IEEE/ACM 28th International Conference on Automated Software Engineering (ASE '13)*, pp. 646–651, IEEE, Silicon Valley, Calif, USA, November 2013.

[8] M. Linares-Vásquez, C. McMillan, D. Poshyvanyk, and M. Grechanik, "On using machine learning to automatically classify software applications into domain categories," *Empirical Software Engineering*, vol. 19, no. 3, pp. 582–618, 2014.

[9] N. Sawadsky, G. C. Murphy, and R. Jiresal, "Reverb: recommending code-related web pages," in *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*, pp. 812–821, May 2013.

[10] S. Wang, D. Lo, and L. Jiang, "Active code search: incorporating user feedback to improve code search relevance," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*, pp. 677–682, Vasteras, Sweden, September 2014.

[11] S. Wang, D. Lo, B. Vasilescu, and A. Serebrenik, "EnTagRec: an enhanced tag recommendation system for software information sites," in *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME '14)*, pp. 291–300, Victoria, Canada, September 2014.

[12] B. Zhou, X. Xia, D. Lo, C. Tian, and X. Wang, "Towards more accurate content categorization of API discussions," in *Proceedings of the 22nd International Conference on Program Comprehension (ICPC '14)*, pp. 95–105, ACM, Hyderabad, India, June 2014.

[13] W. Chan, H. Cheng, and D. Lo, "Searching connected API subgraph via text phrases," in *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE '12)*, 10 pages, Cary, NC, USA, November 2012.

[14] F. Thung, D. Lo, and J. L. Lawall, "Automated library recommendation," in *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE '13)*, pp. 182–191, Koblenz, Germany, October 2013.

[15] F. Thung, S. Wang, D. Lo, and J. Lawall, "Automatic recommendation of API methods from feature requests," in *Proceedings of the IEEE/ACM 28th International Conference on Automated Software Engineering (ASE '13)*, pp. 290–300, IEEE, Silicon Valley, Calif, USA, November 2013.

[16] W. Shi, X. Sun, B. Li, Y. Duan, and X. Liu, "Using feature-interface graph for automatic interface recommendation: a case study," in *Proceedings of the International Conference on Advanced Cloud and Big Data (CBD '15)*, pp. 296–303, Yangzhou, China, November 2015.

[17] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, no. 4-5, pp. 993–1022, 2003.

[18] M. Borg, P. Runeson, and A. Ardö, "Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1565–1616, 2014.

[19] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using IR methods for labeling source code artifacts: is it worthwhile?" in *Proceedings of the 20th IEEE International Conference on Program Comprehension (ICPC '12)*, pp. 193–202, Passau, Germany, June 2012.

[20] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE '12)*, pp. 70–79, ACM, Essen, Germany, September 2012.

[21] X. Sun, X. Liu, B. Li, Y. Duan, H. Yang, and J. Hu, "Exploring topic models in software engineering data analysis: a survey," in *Proceedings of the 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD '16)*, 2016.

[22] S. W. Thomas, A. E. Hassan, and D. Blostein, "Mining unstructured software repositories," in *Evolving Software Systems*, pp. 139–162, Springer, Berlin, Germany, 2014.

[23] S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein, "Studying software evolution using topic models," *Science of Computer Programming*, vol. 80, pp. 457–479, 2014.

[24] X. Sun, X. Liu, J. Hu, and J. Zhu, "Empirical studies on the NLP techniques for source code data preprocessing," in *Proceedings of the 3rd International Workshop on Evidential Assessment of Software Technologies (EAST '14)*, pp. 32–39, ACM, Nanjing, China, May 2014.

[25] C. J. van Rijsbergen, *Information Retrieval*, Butterworths, London, UK, 1979.

[26] M. Ghafari, C. Ghezzi, A. Mocci, and G. Tamburrelli, "Mining unit tests for code recommendation," in *Proceedings of the 22nd International Conference on Program Comprehension (ICPC '14)*, pp. 142–145, Hyderabad, India, June 2014.

[27] J. Kim, S. Lee, S.-W. Hwang, and S. Kim, "Enriching documents with examples: a corpus mining approach," *ACM Transactions on Information Systems*, vol. 31, no. 1, article 1, 2013.

[28] N. Murakami, H. Masuhara, and T. Aotani, "Code recommendation based on a degree-of-interest model," in *Proceedings of the 4th International Workshop on Recommendation Systems for Software Engineering (RSSE '14)*, pp. 28–29, Hyderabad, India, 2014.

[29] A. Zagalsky, O. Barzilay, and A. Yehudai, "Example overflow: using social media for code recommendation," in *Proceedings of the 3rd International Workshop on Recommendation Systems for Software Engineering (RSSE '12)*, pp. 38–42, IEEE, Zurich, Switzerland, June 2012.

[30] C. Lv, W. Jiang, Y. Liu, and S. Hu, "APISynth: a new graph-based API recommender system," in *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, pp. 596–597, Hyderabad, India, June 2014.

[31] C. Zhang, J. Yang, Y. Zhang et al., "Automatic parameter recommendation for practical API usage," in *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, pp. 826–836, June 2012.

[32] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, pp. 364–374, Zurich, Switzerland, June 2012.

[33] A. Lozano, A. Kellens, and K. Mens, "Mendel: source code recommendation based on a genetic metaphor," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*, pp. 384–387, Lawrence, Kan, USA, November 2011.

[34] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher, "Recommending source code for use in rapid software prototypes," in *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, pp. 848–858, Zurich, Switzerland, June 2012.

[35] N. Murakami and H. Masuhara, "Optimizing a search-based code recommendation system," in *Proceedings of the 3rd International Workshop on Recommendation Systems for Software Engineering (RSSE '12)*, pp. 68–72, Zurich, Switzerland, June 2012.

[36] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings*

*of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 213–222, Amsterdam, The Netherlands, August 2009.

[37] S. Proksch, J. Lerch, and M. Mezini, "Intelligent code completion with bayesian networks," *ACM Transactions on Software Engineering and Methodology*, vol. 25, no. 1, article 3, 2015.

[38] T. Xie and J. Pei, "MAPO: mining API usages from open source repositories," in *Proceedings of the International Workshop on Mining Software Repositories (MSR '06)*, pp. 54–57, Shanghai, China, May 2006.

[39] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: mining and recommending API usage patterns," in *Proceedings of the Object-Oriented Programming, 23rd European Conference (ECOOP '09)*, pp. 318–343, Genoa, Italy, July 2009.

[40] D. S. Eisenberg, J. Stylos, A. Faulring, and B. A. Myers, "Using association metrics to help users navigate API documentation," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '10)*, pp. 23–30, Madrid, Spain, September 2010.

[41] L. Han, M. Qian, X. Xu, C. Fu, and H. Kwisaba, "Malicious code detection model based on behavior association," *Tsinghua Science and Technology*, vol. 19, no. 5, pp. 508–515, 2014.

[42] J. Chen, H. Wang, D. Towey, C. Mao, R. Huang, and Y. Zhan, "Worst-input mutation approach to web services vulnerability testing based on SOAP messages," *Tsinghua Science and Technology*, vol. 19, no. 5, pp. 429–441, 2014.