Enriching API Documentation with Code Samples and Usage Scenarios from Crowd Knowledge

Jingxuan Zhang, He Jiang, Zhilei Ren, Tao Zhang, and Zhiqiu Huang

Abstract—As one key resource to learn Application Programming Interfaces (APIs), a lot of API reference documentation lacks code samples with usage scenarios, thus heavily hindering developers from programming with APIs. Although researchers have investigated how to enrich API documentation with code samples from general code search engines, two main challenges remain to be resolved, including the quality challenge of acquiring high-quality code samples and the mapping challenge of matching code samples to usage scenarios. In this study, we propose a novel approach named ADECK towards enriching API documentation with code samples and corresponding usage scenarios by leveraging crowd knowledge from Stack Overflow, a popular technical Question and Answer (Q&A) website attracting millions of developers. Given an API related Q&A pair, a code sample in the answer is extensively evaluated by developers and targeted towards resolving the question under the specified usage scenario. Hence, ADECK can obtain high-quality code samples and map them to corresponding usage scenarios to address the above challenges. Extensive experiments on the Java SE and Android API documentation show that the number of code-sample-illustrated API types in the ADECK-enriched API documentation is 3.35 and 5.76 times as many as that in the raw API documentation. Meanwhile, the quality of code samples obtained by ADECK is better than that of code samples by the baseline approach eXoaDocs in terms of correctness, conciseness, and usability, e.g., the average correctness values of representative code samples obtained by ADECK and eXoaDocs are 4.26 and 3.28 on a 5-point scale in the enriched Java SE API documentation. In addition, an empirical study investigating the impacts of different types of API documentation on the productivity of developers shows that, compared against the raw and the eXoaDocs-enriched API documentation, the ADECK-enriched API documentation can help developers complete 23.81% and 14.29% more programming tasks and reduce the average completion time by 9.43% and 11.03%.

Index Terms—API Documentation; Code Sample; Usage Scenario; Stack Overflow; Crowd knowledge

1 Introduction

Developers often consult Application Programming Interface (API) reference documentation (API documentation for short) to learn the correct usages of unfamiliar APIs, including functionalities, inheritance relationships, and owned members [1], [2], [3]. However, a lot of API documentation lacks code samples with corresponding usage scenarios (i.e., situations in which specific APIs are called to implement certain functionalities [4]). For example, according to our statistics, only 11% and 6% of API types are illustrated by code samples but with vague usage scenarios in the Java Standard Edition (SE) [5] and Android API documentation [6], respectively. The lack of code samples with usage scenarios hampers developers from programming with APIs. From the perspective of open source community, a survey over more than 2,000 Eclipse and Mozilla developers shows that, almost 80% of participants consider that the lack of code samples with their usage scenarios is an obstacle in understanding APIs [7]. From the perspective of commercial enterprises, two surveys on 698 IBM developers and 1,000 Microsoft developers indicate that, developers highly expect that code samples under different usage scenarios could be provided for as many APIs as possible [8], [9]. Hence, it would be ideal if API documentation can be automatically enriched by code samples with usage scenarios.

In the literature, Kim *et al.* propose a seminal approach named eXoaDocs to automatically enrich API documentation with code samples [10]. Given an API documentation, eXoaDocs first extracts all the API methods and searches them in the Google code search engine. Then, from the top-ranked retrieved webpages, eXoaDocs extracts code samples to build a code sample repository. Next, eXoaDocs characterizes all the code samples with a set of properties so as to aggregate them into groups. Finally, eXoaDocs selects representative code samples from each group and embeds them into the API documentation. Experimental results show that the eXoaDocs-enriched API documentation can boost the productivity of developers. However, there remain the following two challenges to be tackled.

The Quality Challenge. *How to guarantee the quality of the retrieved code samples?* There is no uniform mechanism or specific measure to ensure the quality of the code samples obtained by eXoaDocs, since a code sample from a general code search engine may be written by a junior developer or a beginner without being evaluated by other developers and verified in different conditions [11].

The Mapping Challenge. *How to map code samples to*

Jingxuan Zhang and Zhiqiu Huang were with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics. He Jiang was with the Dalian University of Technology and the Beijing Institute of Technology. Zhilei Ren was with the School of Software, Dalian University of Technology. Tao Zhang was with the College of Computer Science and Technology, Harbin Engineering University. e-mail: (jxzhang@nuaa.edu.cn, jianghe@dlut.edu.cn, zren@dlut.edu.cn, cstzhang@hrbeu.edu.cn, zqhuang@nuaa.edu.cn).

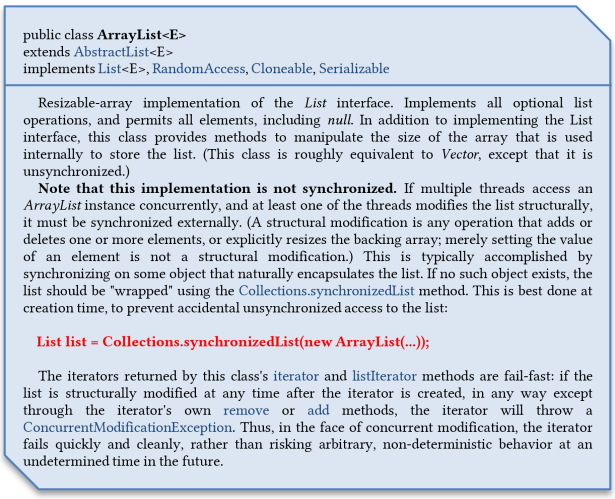
corresponding usage scenarios? There are no explicit mappings between code samples and corresponding usage scenarios in eXoaDocs. Thus, developers have to manually check and test the code samples to clarify their corresponding usage scenarios, which may be time consuming.

To tackle the above challenges, we propose a novel approach named **A**PI **D**ocumentation **E**nrichment with **C**rowd **K**nowledge (**ADECK**). In contrast to eXoaDocs, ADECK leverages crowd knowledge from Stack Overflow, a popular technical Question and Answer (Q&A) website, to identify high-quality code samples and corresponding usage scenarios for enriching API documentation. More specifically, ADECK works as follows. First, it combines questions and their corresponding best answers together in Stack Overflow to achieve a series of Q&A pairs. Then, ADECK extracts APIs from API documentation and employs a traceability linking method to link these APIs with related Q&A pairs. For each linked Q&A pair, ADECK extracts the usage scenario from the question title and its corresponding code sample from the best answer to form a ⟨Usage Scenario, Code Sample⟩ tuple. Next, the tuples linked with each API are clustered and the resulting clusters are ranked by their sizes. Finally, the tuples achieving the highest user score in the top-ranked clusters are embedded into API documentation based on a predefined template.

We evaluate the performance of ADECK on two representative sets of API documentation, namely the Java SE API Documentation (JavaD) in version 7 and the Android API Documentation (AndroidD) in version 4.4. Extensive experiments show that, from the perspective of the code sample quantity, the number of code-sample-illustrated API types in the ADECK-enriched JavaD and AndroidD is 3.35 and 5.76 times as many as that in the raw JavaD and AndroidD. Meanwhile, ADECK achieves true positive rates of 94.83% and 91.73% and improves on eXoaDocs by 26.24% and 8.60%. From the perspective of code-sample quality, the quality of code samples obtained by ADECK statistically outperforms that of code samples by eXoaDocs in terms of *correctness*, *conciseness*, and *usability*. For example, according to the manual evaluations of volunteers, the average *correctness* values of code samples in the ADECK-enriched JavaD and the eXoaDocs-enriched JavaD are 4.26 and 3.28 on a 5-point likert scale, respectively. To evaluate the impacts of different types of API documentation on the productivity of developers, we invite 21 developers to resolve three real programming tasks using JavaD and its enriched versions. Compared against JavaD and the eXoaDocs-enriched JavaD, developers using the ADECK-enriched JavaD are 9.43% and 11.03% faster in terms of the average completion time, and complete 23.81% and 14.29% more programming tasks.

In summary, the main contributions of this study are:

(1) We propose a novel API documentation enrichment approach ADECK. To the best of our knowledge, this is the *first* work to enrich API documentation using code samples with usage scenarios by leveraging crowd knowledge from Stack Overflow.
(2) We conduct extensive experiments to demonstrate the effectiveness of ADECK. Extensive experiments show that ADECK can provide both high-quality code samples and usage scenarios. Meanwhile, the ADECK-enriched API documentation could efficiently boost the productivity of developers.
(3) We open the ADECK-enriched API documentation to the public at https://github.com/APIDocEnrich/ADECK, which may provide new insights into the ways of preparing better API documentation.


public class **ArrayList<E>**
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable

Resizable-array implementation of the *List* interface. Implements all optional list operations, and permits all elements, including *null*. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to *Vector*, except that it is unsynchronized.)

Note that this implementation is not synchronized. If multiple threads access an *ArrayList* instance concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the Collections.synchronizedList method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

List list = Collections.synchronizedList(new ArrayList(...));

The iterators returned by this class's iterator and listIterator methods are fail-fast: if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Fig. 1. *ArrayList* in JavaD

The remainder of the paper is structured as follows. We first show the motivation in Section 2 and the framework of ADECK with its main components in Section 3. Next, we present the experimental setup and experimental results in Section 4 and Section 5. Then, we elaborate the threats to validity and related work in Section 6 and Section 7. Finally, we conclude this study and present future work in Section 8.

2 Motivation

In this section, we first describe the scarcity of code samples in API documentation by illustrating an example. Then, we elaborate why leveraging crowd knowledge in Stack Overflow can address the two challenges, i.e., the quality challenge and the mapping challenge.

2.1 Scarcity of Code Samples in API Documentation

A lot of API documentation is short of high-quality code samples. For example, there are only 11% and 6% of API types illustrated with code samples in JavaD and AndroidD, respectively. As for a specific API *ArrayList* in JavaD, its functional description shown in Fig. 1 illustrates how to instantiate a synchronized *ArrayList* with only one statement [2]. Apart from this statement, there is no code sample illustrating other usages of *ArrayList* in the functional description. The single statement is poorly regulated and far from enough to satisfy developers' various requirements [12]. On the one hand, only one statement is insufficient for developers to correctly program with *ArrayList*, since they do not know how to prepare for it and what are the post operations. On the other hand, developers may encounter various usage scenarios for *ArrayList*, e.g., sorting an *ArrayList*, comparing two *ArrayLists*, and traversing an

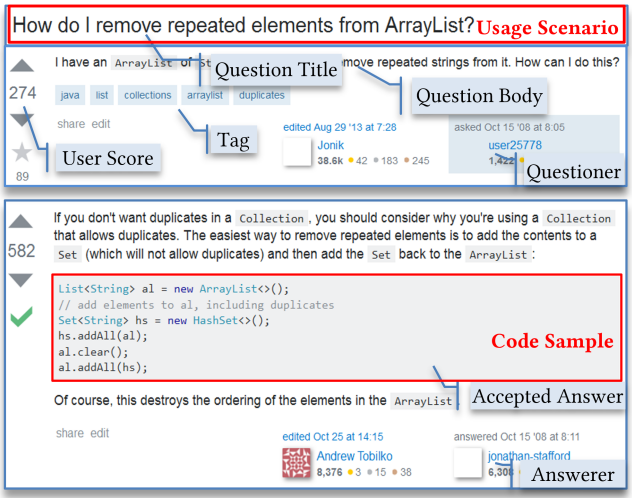


Fig. 2. A Q&A pair example in Stack Overflow

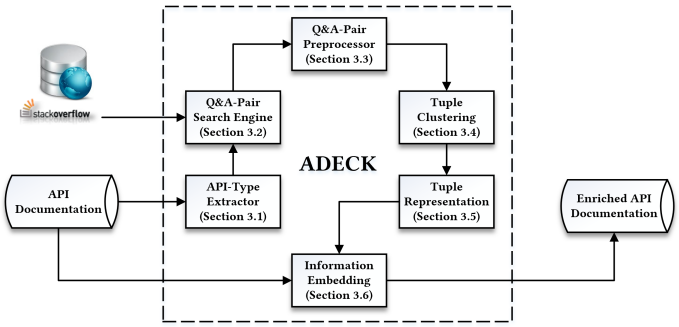


Fig. 3. The framework of ADECK

ArrayList. The existing statement is far from enough to meet the diverse demands of developers [13].

For API documentation, the lack of code samples with corresponding usage scenarios has become a key factor of hampering API understanding [7]. Three surveys conducted on developers from both open source community and commercial enterprises confirm that, developers heavily expect that API documentation provides corresponding code samples under different usage scenarios for APIs as many as possible [7], [8], [9].

2.2 Crowd Knowledge in Stack Overflow

Stack Overflow is a popular technical Question and Answer (Q&A) website with a community of 7.6 million developers and more than 10 million Q&A pairs [14]. Within Stack Overflow, developers often ask a variety of questions about how to use APIs correctly under specific usage scenarios, which are usually described in the question titles. In general, an API related question may attract tens of developers to submit and evaluate answers with code samples. Hence, on the one hand, a high-quality code sample can be acquired from its best answer (the accepted answer or the answer achieving the highest user score). On the other hand, this code sample can be mapped to its corresponding usage scenario in the question title.

Combined with a Q&A pair in Stack Overflow shown in Fig. 2, we illustrate why leveraging crowd knowledge from Stack Overflow could address the two challenges, i.e., the quality challenge and the mapping challenge. There are some items in this Q&A pair, e.g., the question title, the tags, and the user score. *user25778* provides a usage scenario for *ArrayList* in the question title, and *jonathan-stafford* submits the accepted answer with a code sample to resolve it. The accepted answer achieves an extremely high user score, i.e., 582 [15]. It reveals that the code sample in the accepted answer has been evaluated and verified by hundreds of developers in their conditions, so its quality can be well guaranteed. In addition, the code sample in the accepted answer is specifically designed for *ArrayList* under the usage scenario "remove repeated elements from *ArrayList*" in the question title. Hence, the code sample can be credibly mapped to its corresponding usage scenario. In such a way, the two challenges can be tackled by fully leveraging crowd knowledge in Stack Overflow.

3 Framework of ADECK

In this section, we illustrate the framework of ADECK in Fig. 3. The fundamental goal of ADECK is to extract high-quality code samples with their usage scenarios from Stack Overflow and embed them into corresponding API documentation. ADECK takes in Stack Overflow dump files and API documentation as input and generates the ADECK-enriched API documentation. It consists of six components, namely API-Type Extractor, Q&A-Pair Search Engine, Q&A-Pair Preprocessor, Tuple Clustering, Tuple Representation, and Information Embedding. In order to facilitate understanding, we select the *ArrayList* API in the Java SE API documentation as an example to clearly explain how each component works in the following subsections.

3.1 API-Type Extractor

API-Type Extractor identifies and extracts API types from API documentation. We choose APIs at API type level (i.e., class or interface) rather than API member level (i.e., method or field) due to two reasons. On the one hand, the number of usage scenarios for an API member is very limited, since API members may not have complex behaviors [16], [17]. On the other hand, the extracted code samples for API types usually cover the usages of API members [18].

In this paper, we choose JavaD and AndroidD as case studies, since they are well established with different characteristics and attract millions of developers. The selected API documentation is organized as a set of HTML webpages, each of which explains a specific API type in detail with an uniform formatting style [5], [6]. By parsing the title of each webpage, API-Type Extractor can correctly extract the corresponding API type.

As well known, interface APIs do not contain concrete method implementation [15], [19], thus we only extract class APIs and exclude interface APIs. It means that we do not enrich interface APIs with code samples and usage scenarios. To accelerate programming and promote efficiency, developers tend to use non-qualified API names (i.e., simple names) in code samples. Hence, we extract non-qualified API names from API documentation to exactly match APIs

in code samples. The same as Kim *et al.* [10] and Parnin *et al.* [15], if there exist API naming collisions for the same non-qualified name in different packages, we distinguish them by using fully-qualified names. For example, we use "*javax.management.timer.Timer*" and "*java.util.Timer*" to distinguish the two "*Timer*" APIs. Similar to Parnin *et al.* [15] and Bavota *et al.* [20], when extracting class APIs from AndroidD, we only retain the API types inside the "*android.**" packages to avoid repetition with Java SE API types. Finally, we obtain 2,305 Java SE API types and 1,636 Android API types from JavaD and AndroidD, respectively.

3.2 Q&A-Pair Search Engine

Q&A-Pair Search Engine generates a series of Q&A pairs from Stack Overflow dump files and links them to related API types.

Data Collection and Preprocessing. We download Stack Overflow data dump files published on Sep. 8, 2016 [21], and form a series of Q&A pairs by combining each question with its best answer. The same as Nie *et al.* [22], if a question has an accepted answer, we combine it with the accepted answer, since it is well suitable to resolve this question and has been tested and verified by the submitter of this question. If there is no accepted answer for a question, we combine this question with the answer achieving the highest user score, because the other developers agree that this answer can help to resolve this question. The other answers are neglected, since they may convey irrelevant information or provide an inefficient solution. In addition, each Q&A pair is associated with a set of tags, and we only retain the Q&A pairs tagged with "Java" and "Android", respectively [15]. Since we want to obtain code samples for API types, we further filter out those Q&A pairs whose best answers do not contain code sample surrounded by the $\langle pre \rangle$ tag. In such a way, we obtain about 1.2 and 0.9 million Q&A pairs related to Java and Android API types. These Q&A pairs are the sources to obtain high-quality code samples with their usage scenarios.

Traceability Linking. In this study, we employ the traceability linking method proposed by Treude and Robillard [1] to link Q&A pairs to relevant API types. This method consists of two steps. The first step determines a set of candidate Q&A pairs for a specific API type, and the second step links the candidate Q&A pairs to the API type.

To make it easy to understand, an API type is denoted as A and a Q&A pair is denoted as P. The first step of the traceability linking method obtains a set of candidate Q&A pairs $\{P_1, P_2, ..., P_n\}$, which are likely to explain the usages of A. P_i is treated as a candidate Q&A pair, if A appears in the code sample in its best answer. Otherwise, P_i is filtered out. The second step further employs five heuristic rules to link the set of candidate Q&A pairs $\{P_1, P_2, ..., P_n\}$ to A. If P_i in the candidate set matches at least one of the following heuristic rules, it is linked to A. Otherwise, it is filtered out.

(1) Non-qualified A is surrounded by the $\langle code \rangle$ tag in the question body of P_i, e.g., $\langle code \rangle ArrayList \langle /code \rangle$.
(2) Non-qualified A is preceded by "a" or "an" in the question title of P_i, e.g., "How to sort an *ArrayList*".
(3) The question body of P_i has a hyperlink to the official API documentation webpage of A by the $\langle href \rangle$ tag, e.g., $\langle$a href = "https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html"$\rangle ArrayList \langle /a \rangle$.
(4) Non-qualified A is surrounded by punctuations or lower-case words in the question body of P_i, e.g., "Check if a value exists in "*ArrayList*"".
(5) Fully-qualified A appears in the question body or the question title of P_i, e.g., "How to use *java.util.ArrayList*".

Fig. 4. A *Debug-corrective* Q&A pair example

Experimental results show that these heuristic rules are effective to link Q&A pairs to API types [1]. For example, employing the five heuristic rules, the F-Measure values of linking Q&A pairs to one-word API types (e.g., *List*) and multi-word API types (e.g., *ArrayList*) are 90% and 94%, respectively [1]. By applying these heuristic rules to the candidate Q&A pairs, we can obtain the linked Q&A pairs for a specific API type.

Example. Taking the *ArrayList* API from JavaD as an example, after identifying candidate Q&A pairs and applying the five heuristic rules, Q&A-Pair Search Engine links 5,023 Q&A pairs to it based on this traceability linking method.

3.3 Q&A-Pair Preprocessor

Q&A-Pair Preprocessor extracts usage scenarios and code samples from the linked Q&A pairs for each API type, and combines them into ⟨Usage Scenario, Code Sample⟩ tuples. To deal with the mapping challenge, we map the code samples with their usage scenarios in the same Q&A pairs. In such a way, the code samples are exactly used under the corresponding usage scenarios.

Usage Scenario Extraction. A usage scenario should be as concise as possible so that developers can quickly understand its functionality. Similar studies regard the question title as the intent of the code sample in the best answer in Stack Overflow [23], since the question title is the

straightforward target and summary of a Q&A pair. Hence, in this study, we similarly regard the question title of a Q&A pair as its usage scenario.

Code Sample Extraction. Code samples are typically surrounded by the ⟨pre⟩ tag in Stack Overflow, hence it is easy to extract the code sample from the best answer of a Q&A pair. If there is more than one code sample in the best answer, we merge them together to generate a complete code sample as done by Ghafari *et al.* [24]. The extracted usage scenario and code sample shown in the red rectangles in Fig. 2 give an example of the extraction procedure.

As stated by Parnin *et al.* [15] and de Souze *et al.* [25], there are five types of Q&A pairs, namely *How-to-do-it*, *Conceptual*, *Seeking-something*, *Debug-corrective*, and *Miscellaneous*. *How-to-do-it* Q&A pairs provide scenarios in the questions and seek for their implementation. *Conceptual* Q&A pairs seek for the explanation of particular topics or discussion on certain operations. *Seek-something* Q&A pairs seek for something related to programming. *Debug-corrective* Q&A pairs seek for correct code samples for given problematic ones. In addition, *Miscellaneous* Q&A pairs have multiple purposes. For example, a *Miscellaneous* Q&A pair may want to figure out the definition of a specific topic and furthermore seek for its implementation. The five types of Q&A pairs can be handled by following the above procedures except for the *Debug-corrective* Q&A pairs, which need special processing. In a *Debug-corrective* Q&A pair, the questioner usually provides a buggy code sample with a runtime error or an exception. The other answerers try to locate and fix the bug by providing a patch. If we extract usage scenarios and code samples using the above method, the usage scenarios will contain the keywords like "error" and code samples are incomplete with only patches. As a result, *Debug-corrective* Q&A pairs need special treatment after they are identified.

***Debug-corrective* Q&A Pairs Identification.** Based on our observation on about 500 *Debug-corrective* Q&A pairs, we find that they tend to have the following two characteristics in common:

(1) The question title of a *Debug-corrective* Q&A pair often employs some keywords, i.e., "error", "exception", "fail", and "issue", to describe the problem in practice.
(2) The question body of a *Debug-corrective* Q&A pair usually contains a stack trace capturing the runtime error or exception to help the other developers debug.

The first characteristic can be captured by keyword matching and the second one can be matched by some regular expressions defined by Linares-Vsquez *et al.* [26]. If a Q&A pair conforms to any of the two characteristics, ADECK regards it as a *Debug-corrective* Q&A pair. According to our statistics, more than 18% Q&A pairs are identified as the *Debug-corrective* Q&A pairs among the linked Q&A pairs, and this percentage of the Debug-corrective Q&A pairs is similar with the percentage reported by Nasehi *et al.* [27]. In addition, we also randomly sample 100 identified *Debug-corrective* Q&A pairs to manually check whether they really handle the code problems in development. As a result, we find that 91 Q&A pairs really deal with the code problems. Hence, the precision of identifying the *Debug-correct* Q&A pairs is 91%.

Fig. 4 shows a Q&A pair in Stack Overflow [28]. We can see that this Q&A pair follows both the two *Debug-corrective* characteristics, since the question title contains the keyword "error" and the question body provides a runtime error stack trace. Taking the *Debug-corrective* Q&A pair in Fig. 4 as an example, we illustrate the special processing procedures.

Usage Scenario Extraction for *Debug-corrective* Q&A Pairs. If there are Debug-corrective keywords, we remove them with their prepositions from the question title, and treat the processed title as the usage scenario. Hence, the usage scenario of the Q&A pair in Fig. 4 is "write *ArrayList* String to File" after removing the keyword "Error" with the preposition "in".

Bug-Free Code Sample Extraction for *Debug-corrective* Q&A Pairs. We combine the buggy code sample and its patch together to generate a bug-free code sample. This procedure consists of three steps, i.e., buggy code and patch extraction, bug location detection, and bug fix.

In buggy code and patch extraction, we obtain the buggy code and the corresponding patch from the question body and the best answer of a *Debug-corrective* Q&A pair, respectively. Buggy codes are identified from the question body by the ⟨pre⟩ tag, except for the runtime error stack trace, which can be captured by regular expressions [26]. In addition, if there is only one code sample in the best answer, ADECK treats it as the patch. However, if there is more than one code sample, we first employ keyword matching to identify the real patch, since developers tend to use a set of keywords to express the comparison relationship, i.e., "change...to...", "instead of", and "rather than" [29]. If these keywords exist, we distinguish the patch based on the meanings of these keywords from the best answer. For example, the code sample after "to" is identified as the patch, if the keywords "change...to..." are matched. If there is no keyword matched, we combine all the code samples in the best answer and regard them as the patch.

In bug-location detection, we identify the location where the bug exists. We split the buggy code sample and the patch into statements, and only retain valid statements with their sequences by removing blank lines and punctuation lines. Then, similar as Gao *et al.* [29], edit distance is calculated between each statement in the buggy code sample and each statement in the patch. One statement is considered to be transformed from another statement, if their edit distance is less than 0.1. In such a way, we can locate the first buggy statement and the last buggy statement, both of which have the edit distance lower than 0.1 with a statement in the patch. After detecting the first buggy statement and the last buggy statement, we can locate the buggy code block.

In bug fix, we replace the located buggy code block with the patch, and keep the other statements the same. In such a way, we can obtain the complete bug-free code sample.

Example. From the example shown in Fig. 4, we can distinguish the buggy code from the runtime error stack trace. Meanwhile, the unique code sample in the best answer is treated as the patch. After detecting the bug location, we can identify the buggy code block shown in the red rectangle. Finally, we replace the buggy code block with the patch to generate a bug-free code sample. In such a way, we can extract usage scenarios and bug-free code samples from *Debug-corrective* Q&A pairs.

3.4 Tuple Clustering

Tuple Clustering clusters ⟨Usage Scenario, Code Sample⟩ tuples for each API type to avoid duplicate tuples.

Similarity Calculation. Before clustering related tuples for each API type, we should first define the similarity between two ⟨Usage Scenario, Code Sample⟩ tuples. Inspired by the method proposed by Higo and Kusumoto [30], the similarity consists of three parts, namely, Usage scenario Lexical Similarity (ULS), Code sample Lexical Similarity (CLS), and Code sample Structural Similarity (CSS). The three parts of this similarity measure different aspects of code samples, and functional sameness between code samples can be detected by combining them together [30].

ULS and CLS are measured by the widely used cosine similarity [31]. Each usage scenario or code sample is represented as a vector after a series of Natural Language Processing (NLP) steps, namely tokenization, stemming, and stop word removal [31]. In the vector, each dimension stands for a token and its value stands for the weight of this token. The weights of tokens are calculated by the widely used Term Frequency (TF) × Inverse Document Frequency (IDF) weighting scheme [16]. Then, cosine similarity is calculated to measure the cosine of the angle between the two vectors in the inner product space.

CSS measures API-call-sequence similarity at the character level, and similar studies have shown its effectiveness [30], [32]. CSS is calculated as follows. First, the variable names and method names in code samples are replaced with the strings of "variable" and "method" respectively to eliminate the influence of different names. Then, each code sample is converted into a token sequence, in which all the spaces are deleted. Next, the Longest Common Subsequence (LCS) is identified between the two token sequences. Finally, CSS is defined as the length of LCS divided by the length of the shorter token sequence.

$$CSS(C1, C2) = \frac{|LCS(S1, S2)|}{min\{Len(S1), Len(S2)\}} \quad (1)$$

where $S1$ and $S2$ are two token sequences derived from two code samples $C1$ and $C2$, $LCS(S1, S2)$ is the LCS for $S1$ and $S2$. $Len(Si)$ is the length of Si (i = 1 or 2).

The three similarities should be normalized before combining them together, i.e., each value is divided by the maximal value in each similarity for each API type. In this study, we define that the three similarities contribute equally to the final score. Hence, the final similarity score between two tuples can be calculated as follows:

$$Simi(T1, T2) = \frac{\overline{ULS}(U1, U2) + \overline{CLS}(C1, C2) + \overline{CSS}(C1, C2)}{3} \quad (2)$$

where $T1 = \langle U1, C1\rangle$ and $T2 = \langle U2, C2\rangle$ represent two distinct ⟨Usage Scenario, Code Sample⟩ tuples. $\overline{X}$ means the normalized value of X, e.g., $\overline{ULS}$ is the normalized ULS.

Clustering. Based on the similarity of every two ⟨Usage Scenario, Code Sample⟩ tuples, we can obtain the similarity matrix for all the linked tuples of an API type. The similarity matrix is used as the input of the clustering algorithm. In this study, we employ the APCluster algorithm using affinity propagation to cluster ⟨Usage Scenario, Code Sample⟩ tuples, since APCluster can automatically determine the optimal value of the cluster number [33].

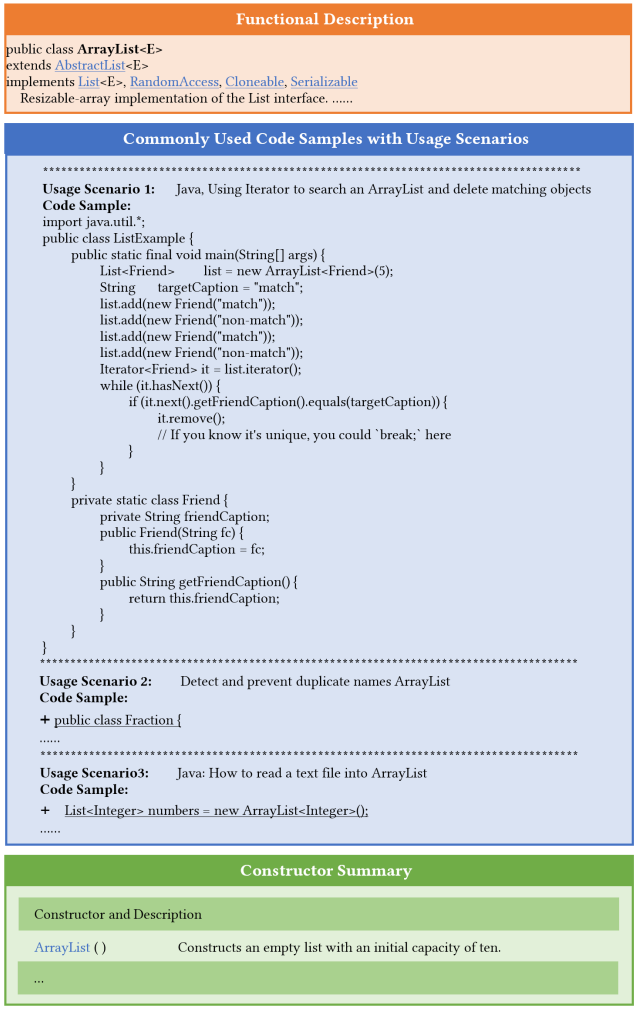


Fig. 5. The enriched API documentation for *ArrayList*

Example. We obtain 388 clusters after clustering related tuples for *ArrayList* by applying APCluster. We can see that developers will encounter various usages of *ArrayList* in practice. The biggest cluster containing 233 tuples illustrates the usage scenario "searching a specified object in an *ArrayList*". It shows that this usage scenario is commonly discussed by developers.

3.5 Tuple Representation

Tuple Representation selects the most representative ⟨Usage Scenario, Code Sample⟩ tuple from each resulting cluster. To deal with the quality challenge, we employ the user score voted by the crowds in Stack Overflow as the indicator of the quality. We only select those ⟨usage scenario, code sample⟩ tuples achieving the highest user scores in clusters as representatives and embed them into API documentation.

The ⟨Usage Scenario, Code Sample⟩ tuples in each cluster are assumed to illustrate the same usage scenario for an API type. Similar to the method proposed by Nasehi *et al.* [27], the tuple achieving the highest user score is chosen to represent each cluster. In such a way, the quality of the extracted code sample can be guaranteed.

The selected representative tuples can better cover the API usage scenarios that are most desired by developers. The more tuples in a cluster, the more discussions around it by developers. Hence, the number of tuples in a cluster could reflect the popularity of its corresponding usage scenario. We rank the clusters by their sizes (the number of tuples in them) in a descending order, and recommend the representative tuples in the top 10 clusters for each API type. Recommending the top 10 results is a common practice in recommendation systems within software engineering, and many studies also evaluate their results based on the top 10 results [25], [29]. If the number of automatically generated clusters is less than 10, we recommend all the representative tuples.

Example. Still taking *ArrayList* as an example, the top 10 largest clusters include 1486 tuples in total, which covers nearly 30% of the linked Q&A pairs. The average user score of the top 10 tuples for *ArrayList* is 63, which is relatively high [27]. It means that these representative tuples have been checked and verified by many developers in their conditions. Hence, the quality of these representative tuples is likely to be high.

3.6 Information Embedding

Information Embedding embeds the top-ranked representative ⟨Usage Scenario, Code Sample⟩ tuples into API documentation. We define a template to organize these tuples for API types, and place them between the functional description and the constructor or method summary table in their API-documentation webpages. Since ⟨Usage Scenario, Code Sample⟩ tuples are relatively independent from the existing information in API documentation, no matter where we place them, they will not decrease the intelligibility and readability of the entire API documentation [34]. By reformatting the webpages of API documentation, the ADECK-enriched API documentation can be generated.

Example. Fig. 5 shows an example of the ADECK-enriched API documentation for *ArrayList* with the top 3 usage scenarios, which are placed between the functional description and the constructor summary. Each section includes two parts corresponding to the two elements in each ⟨Usage Scenario, Code Sample⟩ tuple, and the two parts are separated with each other to make them readable and scannable. In such a way, developers can benefit from the enriched API documentation.

4 Experimental Setup

In this section, we try to investigate the following four RQs:

RQ1: What is the quantity of the API types illustrated with code samples and usage scenarios in the ADECK-enriched API documentation?

Motivation. The more code samples with usage scenarios can be enriched in API documentation, the more benefit the enriched API documentation could give to developers. To investigate how many API types are illustrated with code samples and usage scenarios in the ADECK-enriched API documentation, we set up this RQ. In addition, we also would like to investigate the quantities of the code-sample-illustrated API types in the other two types of API documentation, i.e., the raw API documentation and the eXoaDocs-enriched API documentation.

TABLE 1
The selected API types

Id	Java SE	Android
1	java.app.Applet	android.view.View
2	java.awt.Image	android.os.Bundle
3	java.beans.PropertyChangeEvent	android.content.Intent
4	java.io.File	android.app.Activity
5	java.lang.Object	android.content.Context
6	java.net.URL	android.util.Log
7	java.security.Security	android.widget.Textview
8	java.sql.DriverManager	android.view.ViewGroup
9	java.util.ArrayList	android.widget.Button
10	java.swing.Jcomponent	android.view.LayoutInflater

Method. Specifically, for JavaD and AndroidD, we run ADECK to generate the ADECK-enriched API documentation. Meanwhile, since eXoaDocs is not publicly available, we implement eXoaDocs by ourselves accordingly [10]. We also apply eXoaDocs to generate the eXoaDocs-enriched API documentation. We then count the number of API types illustrated with code samples in the three types of API documentation. In addition, we also identify true positive API types in the enriched API documentation and calculate the true positive rates to further compare eXoaDocs and ADECK. Specifically, three authors of the paper manually classify the code samples in the eXoaDocs-enriched and ADECK-enriched API documentation into false positives and true positives. If a code sample really concentrates on illustrating the proper usages for the linked API type, we judge it as a true positive. Otherwise, we treat it as a false positive. In addition, if all the enriched code samples for an API type are judged as false positives, then the API type is treated as a false positive API type. Otherwise, it is a true positive API type. Each author checks one-third of all the code samples. To guarantee the quality of the evaluation, the code samples evaluated by an author are double checked by another author. If there exists a disagreement, the two authors sit together and discuss it to reach an agreement. The validation procedure takes each author about one week to complete.

RQ2: What is the quality of the code samples in the ADECK-enriched API documentation?

Motivation. The enriched code samples with usage scenarios are expected to have a high quality. In this RQ, we would like to investigate the quality of the code samples in the ADECK-enriched API documentation, and whether it is superior to the quality of the code samples in the eXoaDocs-enriched API documentation.

Method. Similar to Treude *et al.* [1] and Beyer *et al.* [35] and suggested by the Apatite tool [36], we select the code samples of the top 10 most commonly used API types shown in Table 1 to evaluate, since these API types have a broad coverage to resolve common programming tasks. All these API types can be enriched with code samples by both eXoaDocs and ADECK. In addition, to achieve a credible evaluation, we invite the undergraduate students who achieve an examination score higher than 90/100 in the

Advanced Java Programming class from Dalian University of Technology. Eventually, 8 graduate students agree to participate in this experiment. All these volunteers are familiar with Java and Android development with more than four years of experience in programming, so they can qualify for this evaluation.

We extract the enriched code samples for the selected API types from the eXoaDocs-enriched and ADECK-enriched API documentation, respectively. It should be noted that we only extract code samples without corresponding usage scenarios from the ADECK-enriched API documentation so that we can make a fair comparison. We distribute each code sample to two different volunteers to evaluate and score, and their average value is treated as the final score. To avoid the evaluation bias, the origins (eXoaDocs or ADECK) of the code samples are unknown to the volunteers. Similar as Kim *et al.* [10], after reading and testing the assigned code samples, each volunteer is required to grade each assigned code sample independently based on three quality evaluation criteria:

(1) *Correctness* - Are the code samples correct without an error?
(2) *Conciseness* - Do the code samples contain mandatory steps to implement a functionality with less statements?
(3) *Usability* - Are the code samples easy to understand and reuse?

The score of each criterion is on a 5 points likert scale [10], whose guidelines are as follows: {1-very low, 2-low, 3-moderate, 4-high, 5-very high}. A higher score represents a higher quality of a criterion for the enriched code samples. Meanwhile, we also calculate the Weighted Kappa Agreement for volunteers on evaluating the quality of the code samples to show their agreements.

RQ3: What are the coincidence levels between code samples and their corresponding usage scenarios in the ADECK-enriched API documentation?

Motivation. Different from the eXoaDocs-enriched API documentation containing only code samples for API types, the ADECK-enriched API documentation includes both code samples and their usage scenarios. In this RQ, we would like to investigate whether the code samples are consistent with their corresponding usage scenarios in the ADECK-enriched API documentation.

Method. We select the same API types as RQ2 to evaluate. We extract both the code samples and their corresponding usage scenarios for the selected API types from the ADECK-enriched API documentation. The same volunteers as RQ2 are required to score all the ⟨Usage Scenario, Code Sample⟩ tuples for the selected API types. Before the annotation, each volunteer is assigned with an experimental tutorial. In this experimental tutorial, we present the annotation procedures, the annotation criterion, some matters needing attention, and a running example. The scoring criterion of the coincidence level is *Consistency* - Do the usage scenarios describe the real functionalities of their corresponding code samples? After they fully understand this experimental tutorial, they are required to perform the annotation. All the ⟨usage scenario, code sample⟩ tuples for the selected API types are assigned to the volunteers sequentially, and the volunteers independently evaluate and score the coincidence levels between them. Similarly, each tuple is scored by two volunteers independently on a likert scale of 5 points [10], and the final score is obtained by averaging the two individual scores. In addition, the Weighted Kappa Agreement is also calculated to show the agreements between volunteers.

RQ4: Can the ADECK-enriched API documentation boost the productivity of developers?

Motivation. To investigate whether the ADECK-enriched API documentation is helpful for developers to resolve programming tasks, we set up this RQ.

Method. First, we invite 21 developers to participate in the experiment, including 6 Ph.D. candidates and 15 master students from Dalian University of Technology. Before conducting the experiment, the developers are required to complete a survey with four questions [37] to investigate their time to start programming, their frequencies of referring to API documentation, their adept programming languages, and their proficiency levels of Java SE APIs and Android APIs. Based on their responses to the survey, we divide the developers into three groups with similar or equivalent programming skills.

Then, we provide the developers with an experimental guideline introducing the related concepts and workflow to help them understand the overall procedures of the experiment. The developers are required to fully understand the guideline before conducting the experiment. To motivate the developers to attentively accomplish these programming tasks, we set up an incentive mechanism. Developers will win a prize (i.e., an USB flash disk), if they perform well in completing the tasks.

Next, inspired by Kim *et al.* [10], Xie *et al.* [38], and Lin *et al.* [39], we design three programming tasks according to the following three criteria.

(1) The goals of the programming tasks should be intuitive and easy to understand without ambiguity, so developers could focus on resolving the programming tasks rather than comprehending them.
(2) The programming tasks should have different levels of difficulties to distinguish different types of API documentation.
(3) The programming tasks should be related to different programming topics, so they can match different capabilities of developers.

The same as Kim *et al.* [10], we follow three steps to determine the programming tasks. First, we select the API types which are illustrated by code samples in the eXoaDocs-enriched and ADECK-enriched API documentation but not in the raw API documentation, thus we can know whether the enriched API documentation is effective. In addition, we can also compare the eXoaDocs-enriched API documentation and the ADECK-enriched API documentation. Second, we search these API types in Java tutorials and blogs to find their related programming tasks. Third, the authors of this paper conduct some preliminary programming tasks to decide whether they are suitable as the final programming tasks. Through an open discussion, we eventually determine the three programming tasks under the three selection criteria.

The programming tasks are shown in Table 2. The first programming task is easy and related to string processing, which tries to split a string based on a specified character.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2919304, IEEE Transactions on Software Engineering

TABLE 2
The designed programming tasks

Id		1	2	3
Programming task		Split a text based on a specified character	Read and print the source code of a webpage	List the first menu when pressing the Ctrl key
Related topic		String processing	Network connection and interaction	GUI design and implementation
Potentially Useful API		java.util.String.Tokenizer	java.net.URL.Connection	javax.swing.JMenuBar
Test case	Test Input	class1=1\nclass2=2\nclass3=3\nclass4=4 \nclass1=5\nclass2=6\nclass3=7\nclass4=8 \nclass1=9\nclass2=10\nclass3=11\nclass4=12	http://global.bing.com/?FORM=HPCNEN &setmkt= en-us&setlang=en-us	Run the program and press the Ctrl key
	Expected Output	class1: 1, 5, 9 class2: 2, 6, 10 class3: 3, 7, 11 class4:4, 8, 12	The source code of bing search	The menus in the first menu bar are shown
Level of Difficulty		easy	moderate	hard

The second programming task aims to obtain the source code of a webpage, which is a moderate one related to the network connection. The third programming task is about GUI design to list the menus when receiving a keyboard press. These programming tasks are coupled with some potentially useful APIs. In addition, the same as Kim *et al.* [10], we also provide a test case related to each programming task for developers to verify whether a programming task is correctly completed or not.

We follow the crossover design procedures to conduct the experiment [40]. As shown in Table 3, developers are required to accomplish these programming tasks sequentially with the help of different types of API documentation. For example, the developers in Group 1 (G1) accomplish the first programming task with the help of the raw API documentation, the second programming task with the eXoaDocs-enriched API documentation, and the third programming task with the ADECK-enriched API documentation. In contrast, the developers in Group 2 (G2) and Group 3 (G3) leverage different orders of API documentation to resolve the same three programming tasks. The previous task will not influence the subsequent one, since they are independent and their completion time is calculated separately. The crossover design brings two benefits [40]. On the one hand, the bias of different programming skills of the developers on the experiment can be eliminated. On the other hand, it is statistically efficient and requires fewer developers.

When completing the programming tasks, the developers are not informed which type of API documentation they use. Hence, the perceptions of developers will not have an effect on the results. To better investigate the effectiveness of different types of API documentation in completing programming tasks, during the programming process, the developers are not allowed to look up the answers online. In such a way, they can only rely on the assigned API documentation. The same as Lin *et al.* [39] and Maalej *et al.* [41], we use screen capture tools (i.e., Camtasia Studio in this study [42]) to record all the programming behaviors of developers, since the recorded videos make it easier for us to explicitly analyze the whole programming process compared to the log analyzers [38]. The time to accomplish each task is limited to 30 minutes to avoid endless programming. Developers are not allowed to give up halfway until the programming time reaches to 30 minutes [10].

TABLE 3
The assigned API documentation for each developer group

Group	Programming task 1	Programming task 2	Programming task 3
G1	The raw API documentation	The eXoaDocs enriched API documentation	The ADECK enriched API documentation
G2	The eXoaDocs enriched API documentation	The ADECK enriched API documentation	The raw API documentation
G3	The ADECK enriched API documentation	The raw API documentation	The eXoaDocs enriched API documentation

After the developers complete these programming tasks, we analyze the recorded videos to obtain two evaluation metrics, i.e., the task completion frequency and the average completion time. By comparing the two metrics, we can acquire whether the ADECK-enriched API documentation can boost the productivity of developers.

At last, we conduct an interview with the developers to investigate how the ADECK-enriched API documentation helps them program. After finishing these programming tasks, we explain to the developers why we conduct the interview. We inform the developers the exact programming task in which they use the ADECK-enriched API documentation. According to their programming experience with the ADECK-enriched API documentation, they are asked three questions, i.e., "Whether the ADECK-enriched API documentation is helpful in resolving programming tasks", "If the ADECK-enriched API documentation is helpful, what are the reasons", and "What suggestions would you give to ADECK". The interview is audio-recorded and transcribed for further analysis. The interview could give insights into the rationality and future directions for ADECK.

TABLE 4
Number of API types with code samples

API Documentation		Java SE	Android
# API Types in total		2,305	1,636
The raw API documentation	# API types with code	263	104
The eXoaDocs-enriched API documentation	# API types with code	1,283	812
	# True positive APIs	880	675
	True positive rate	68.59%	83.13%
The ADECK-enriched API documentation	# API types with code	930	653
	# True positive APIs	882	599
	True positive rate	94.83%	91.73%

5 Experimental Results

5.1 Investigation of RQ1

In this RQ, we present the quantity of the API types that can be enriched by ADECK with code samples and usage scenarios.

Result. Table 4 presents the number of API types illustrated with code samples in the three types of API documentation. As shown in Table 4, only 263 and 104 API types are illustrated with code samples in the raw JavaD and AndroidD. In contrast, the number of code-sample-illustrated API types reaches to 1,283 and 812 in the eXoaDocs-enriched JavaD and AndroidD. In addition, 930 API types in the ADECK-enriched JavaD and 653 API types in the ADECK-enriched AndroidD are illustrated by code samples. Hence, the number of API types illustrated by code samples is greatly improved in the eXoaDocs-enriched and ADECK-enriched API documentation.

The manually validated results are also shown in Table 4. We can see that ADECK achieves higher true positive rates than eXoaDocs. For example, 880 and 675 API types are identified as true positive API types, and the true positive rates are 68.59% and 83.13% in the eXoaDocs-enriched JavaD and AndroidD, respectively. In contrast, the ADECK-enriched JavaD and AndroidD contain 882 and 599 true positive API types, and their proportions account for 94.83% and 91.73%, respectively. The number of true positive API types in the ADECK-enriched JavaD and AndroidD are 3.35 and 5.76 times as many as the number of API types with code samples in the raw JavaD and AndroidD.

We can see that eXoaDocs achieves a higher number of true positive API types than ADECK, but the disparity is not very big. For example, ADECK obtains two more true positive API types than eXoaDocs in the enriched JavaD. When considering AndroidD, there are 675 and 599 true positive API types in the eXoaDocs-enriched AndroidD and ADECK-enriched AndroidD, respectively. It means that eXoaDocs achieves a higher Recall value than ADECK. What is more, ADECK achieves higher true positive rates than eXoaDocs. The reason may be that eXoaDocs only relies on simple lexical matching by searching general code search engines. As a result, even though plentiful code samples can be obtained, these code samples often contain the steps to run and configure a project or some command lines with logs recording the runtime information, which are irrelevant with API types. In contrast, ADECK employs an accurate traceability linking method to link code samples with API types. Hence, the enriched code samples of ADECK are highly likely to focus on API usages.

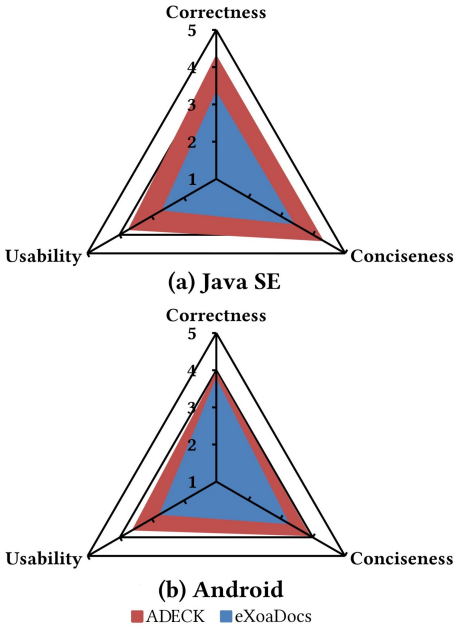


Fig. 6. Evaluation scores for Java SE and Android

Furthermore, we also investigate the quantity of true positive API types by combining eXoaDocs and ADECK. Combining both eXoaDocs and ADECK to enrich API documentation, the number of true positive API types can be further improved, i.e., 1,209 for Java SE and 894 for Android. It means that more than 50% API types can be truly covered. Hence, eXoaDocs and ADECK can cooperate and complement each other to better enrich API documentation.

Conclusion. The number of code-sample-illustrated API types is greatly improved in the ADECK-enriched API documentation compared against the raw API documentation. In addition, ADECK achieves higher true positive rates than eXoaDocs.

5.2 Investigation of RQ2

In this subsection, we present the evaluation results of the quality of the enriched code samples.

Result. Fig. 6 presents the average evaluation scores on the quality of the code samples in the eXoaDocs-enriched and ADECK-enriched API documentation. We can see that the quality of code samples in the ADECK-enriched API documentation is higher than that of code samples in the eXoaDocs-enriched API documentation in terms of *correctness*, *conciseness*, and *usability*. For example, the average scores of the code samples in the ADECK-enriched JavaD and AndroidD are 4.26 and 3.9 in terms of *correctness*. In contrast, the values in the eXoaDocs-enriched JavaD and AndroidD are only 3.28 and 3.71, respectively. In terms of *conciseness* and *usability*, the code samples in the ADECK-enriched API documentation also achieve higher average scores than those code samples in the eXoaDocs-enriched API documentation. In addition, the Weighted

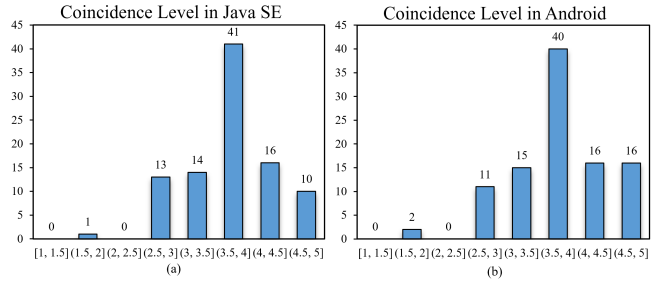


Fig. 7. Frequency histograms of coincidence level

Kappa Agreement between volunteers on evaluating code samples is 0.57 showing a moderate agreement.

Moreover, we conduct paired Wilcoxon signed rank test to explore the statistical significance of the difference between the quality of the code samples in the eXoaDocs-enriched API documentation and that of code samples in the ADECK-enriched API documentation. We define the following hypothesis:

H_0: There is no significant difference between the quality of the code samples in the eXoaDocs-enriched API documentation and that of code samples in the ADECK-enriched API documentation.

In this study, the significance level is set to 5%. If the p-value is less than the significance level, H_0 is rejected and there is a significant difference between the quality of the code samples in the eXoaDocs-enriched API documentation and that of code samples in the ADECK-enriched API documentation. The p-values are {3.845e-08, 8.983e-09, and 7.135e-09} for Java SE and {0.04876, 5.933e-09, and 2.103e-09} for Android in terms of *correctness*, *conciseness*, and *usability*. All the p-values are lower than 0.05. Hence, H_0 is rejected. It means that there is a significant difference between the quality of the code samples in the eXoaDocs-enriched API documentation and that of the code samples in the ADECK-enriched API documentation.

The reason why ADECK achieves code samples with a higher quality than eXoaDocs may be that, ADECK combines questions with their best answers in Stack Overflow and selects the ⟨Usage Scenario, Code Sample⟩ tuples with the highest user scores in the top-ranked clusters, thus the quality of code samples is inevitably high. In contrast, eXoaDocs obtains code samples from general code search engine without a similar mechanism to guarantee the quality of code samples.

Conclusion. There is a statistical difference between the quality of the code samples in the eXoaDocs-enriched and ADECK-enriched API documentation. ADECK achieves code samples with higher quality than eXoaDocs.

5.3 Investigation of RQ3

In this RQ, we present the evaluation results of the coincidence level between code samples and usage scenarios in the ADECK-enriched API documentation.

Result. Fig. 7 shows the frequency histograms for the scores with respect to the coincidence levels between usage scenarios and code samples in the ADECK-enriched Java SE and Android API documentation. Considering that the eXoaDocs-enriched API documentation does not contain usage scenarios, we only evaluate the ADECK-enriched API documentation without comparison. As seen from the figure, the distributions of the scores show inverted U shape curves on the whole. The scores of the coincidence level are distributed between 1.5 and 5. No less than 40 scores range from 3.5 to 4 in both the ADECK-enriched JavaD and AndroidD, and this range takes up the largest percentage. Whether the scope becomes either larger or smaller, the corresponding number gradually decreases. The average scores are 3.96 and 4.02 on a 5 points likert scale in the ADECK-enriched JavaD and AndroidD, respectively. It means that the code samples are relatively consistent with the corresponding usage scenarios. The reason may be that, a code sample and its corresponding usage scenario are extracted from the same Q&A pair, in which the code sample is specially designed for the usage scenario. In addition, the Weighted Kappa Agreement between volunteers on evaluating the coincidence level is 0.50 showing a moderate agreement.

Conclusion. The code samples are relatively consistent with their corresponding usage scenarios in the ADECK-enriched API documentation.

5.4 Investigation of RQ4

In this RQ, we compare different programming tasks, developer groups, and API documentation in terms of the task completion frequency and the average completion time. In addition, similar as RQ2, we also conduct the paired Wilcoxon signed rank test to explore whether the programming tasks, the developer groups, and different types of API documentation are statistically different.

5.4.1 Comparison between Different Tasks

Result. Fig. 8 shows the comparison results between different programming tasks. In addition, we also present asterisks and brackets on the bar chart (showing the task completion frequency) and the line chart (showing the completion time) to indicate there exits statistically significant difference between the two corresponding columns. We can see that the completion frequency of Task 1 is the largest, i.e., 19. Furthermore, the average completion time of Task 1 is the shortest (16.99 minutes) among all the three tasks. Hence, it is coincident with Table 2 showing that Task 1 is the easiest one to complete. In contrast, the completion frequency of Task 3 is the smallest, i.e., 10, and its average completion time is the longest, i.e., 23.58 minutes. It means that Task 3 is the hardest one. In addition, since the average completion time is less than 25 minutes for all the tasks, the 30 minutes completion time limit is within a feasible and reasonable range.

Fig. 8 also presents the results of the paired Wilcoxon signed rank test. We can find that there are statistically significant differences between Task 1 and Task 2 and between Task 1 and Task 3 in terms of the task completion frequency. In addition, from the perspective of the task completion time, each pair of the three tasks is significantly different.

Conclusion. The task completion frequency and the average completion time are consistent with the levels of difficulties of the programming tasks.

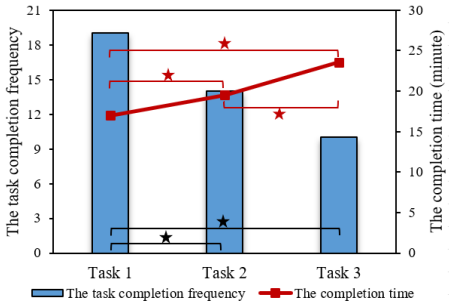


Fig. 8. Comparison between different tasks

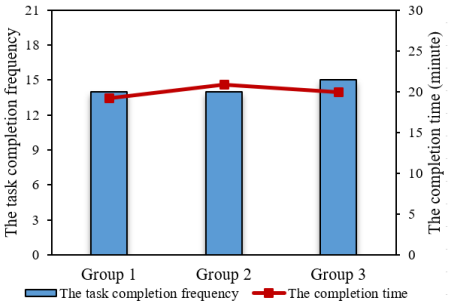


Fig. 9. Comparison between different groups

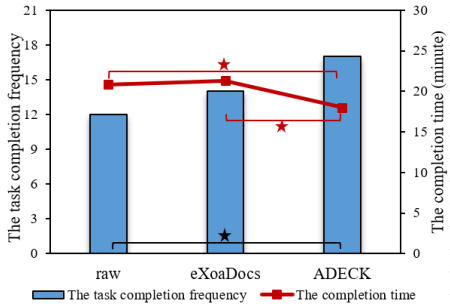


Fig. 10. Comparison between different API documentation

5.4.2 Comparison between Different Groups

Result. Fig. 9 shows the comparison results between different developer groups in terms of the task completion frequency and the average completion time. The task completion frequencies of the three groups are similar, i.e., 14, 14, and 15, respectively. From the perspective of the average completion time, the three groups take 19.25, 20.88, and 19.97 minutes, respectively, in which the divergence is less than 2 minutes. That is to say, similar results are achieved in the three developer groups, which confirms the validity of our grouping strategy. This is because before conducting the experiment, we carry out a survey to investigate their programming skills. Through the survey, we can know their proficiency levels in Java SE APIs so as to divide them into groups with similar programming skills.

We also investigate whether there is statistically significant difference between each pair of the developer groups in terms of the task completion frequency and the completion time. We find that all of the p-values are larger than 0.05, which means that the three developer groups are not significantly different. Hence, there is no asterisk and bracket to indicate the difference between the developer groups in Fig. 9.

Conclusion. The developers are relatively evenly distributed into groups with similar programming skills.

5.4.3 Comparison between Different Types of API Documentation

Result. Fig. 10 shows the comparison results between different types of API documentation. Similarly, we employ asterisks and brackets to indicate whether there are significant differences between different types of API documentation. We can see that the task completion frequency of the raw API documentation is the least, i.e., 12. In contrast, the task completion frequencies of the eXoaDocs-enriched and ADECK-enriched API documentation are 14 and 17, respectively. In addition, the average completion time of the eXoaDocs-enriched API documentation is the longest. i.e., 21.30 minutes. Whereas, the average completion time of the raw API documentation and the ADECK-enriched API documentation are 20.82 minutes and 17.99 minutes, respectively. The disparity of the average completion time between the raw API documentation and the eXoaDocs-enriched API documentation is trivial, i.e., 0.48 minute. By examining the videos recorded by the screen capture tool, we find that developers tend to spend more time on reading and changing the code samples given in the eXoaDocs-enriched API documentation, which may prolong the completion time.

When focusing on the eXoaDocs-enriched and ADECK-enriched API documentation, we note that developers are able to accomplish more programming tasks within less average time with the help of the ADECK-enriched API documentation than utilizing the eXoaDocs-enriched API documentation. For instance, the task completion frequency of the ADECK-enriched API documentation is 17, whereas it is only 14 for the eXoaDocs-enriched API documentation. It means that additional 3 developers (14.29% in the invited 21 developers) are successfully helped. Furthermore, the average completion time of the ADECK-enriched API documentation is 17.99 minutes, while it is 21.30 minutes for the eXoaDocs-enriched API documentation. It implies that with the help of the ADECK-enriched API documentation, developers shorten the completion time by 11.03% in a time frame of 30 minutes compared to the eXoaDocs-enriched API documentation.

In terms of the results of the paired Wilcoxon signed rank test, we can find that the developers with the raw API documentation are statistically different with the developers with the ADECK-enriched API documentation in terms of the task completion frequency and the completion time. In addition, the developers with the eXoaDocs-enriched API documentation are significantly different with the developers with the ADECK-enriched API documentation in terms of the completion time. Considering that the developers using the ADECK-enriched API documentation can complete more programming tasks within less time than those using the raw API documentation and those using the eXoaDocs-enriched API documentation, we can say that leveraging the ADECK-enriched API documentation could boost the productivity of developers.

Conclusion. Using the ADECK-enriched API documentation, developers can complete more programming tasks with less time than using the eXoaDocs-enriched API documentation.

5.4.4 Post Study Interview

We interview developers with three questions after they finish their programming tasks. We transcribe the interview audio record into text. To better understand the reasons and the suggestions from the developers, we conduct a thematic analysis on the interview responses. The thematic analysis tries to identify themes or patterns by grouping the responses. As described by Coelho *et al.* [43] and Silva

et al. [44], the thematic analysis consists of four steps: (1) scanning and getting familiar with the responses. (2) generating initial themes from the responses. (3) reviewing and refining the generated themes. (4) defining the final themes. The thematic analysis is conducted by two authors. If there is a conflict between the two authors in generating the final themes, they discuss it to reach an agreement.

For the first question, all the 21 developers (100%) confirm that the ADECK-enriched API documentation is helpful for resolving programming tasks. For the second question related to the reasons why ADECK works, developers provide their comments. We obtain three reasons why the ADECK-enriched API documentation works. (1) The maps between code samples and usage scenarios are helpful. 12 developers agree with this reason. As one developer mentions, "*ADECK adds additional code samples into API documentation. Most importantly, the code samples are mapped to their usage scenarios. They are helpful for resolving some programming tasks*". (2) The organization is clear. 5 developers confirm this reason. As one developer says, "*The organization of the enriched API documentation is clear. It is convenient to point out the usage scenarios for code samples*". (3) The quality of the code samples is relatively high. 4 developers agree with this reason. As a developer states, "*The quality of the code samples in the enriched API documentation is high. I can easily adapt the code samples to my programming context*". As for the third question, 9 developers provide their suggestions. These suggestions can be roughly divided into two themes. (1) More explanations are needed. 6 developers make such a suggestion. As one developer says, "*Provide more explanations on the code samples*". (2) Pointing out the versions of APIs. 3 developers confirm this suggestion. As a developer mentions, "*It could be better if ADECK can identify the versions of the invoked APIs*". These suggestions motivate us to further improve ADECK in the future.

6 Threats to Validity

6.1 Threats to Internal Validity

First, one potential threat to internal validity is the selection of APIs in RQ2. Sampling different APIs will result in different evaluations of ADECK. To mitigate this threat, we choose the commonly used APIs to evaluate, since they can help developers resolve more programming tasks. Second, one potential threat is the human evaluation. The authors of this paper manually evaluate and identify the true positive APIs in RQ1. We employ the double check mechanism to mitigate this threat. We recruit 8 volunteers to evaluate the quality of code samples and the coincidence level between usage scenarios and code samples in RQ2. To obtain consistent evaluations, we provide some evaluation criteria for the volunteers. In addition, each code sample and the coincidence level are evaluated by two volunteers independently and the final score synthesizes the scores from two volunteers. In such a way, we can reduce the threats as much as possible. We invite 21 developers to participate in RQ3. To avoid introducing biases, we introduce the crossover design to conduct the experiment, which can reduce the influence of different programming skills of the developers. Third, the re-implementation of eXoaDocs may be not exactly the same as the original, which is also a potential threat. We implement eXoaDocs based on the Google search engine rather than the Google code search engine, since the service of the Google code search engine has been discontinued. We employ the code review process to guarantee the quality of the implementation. Finally, the selection of the programming tasks may be another potential threat. In the future, we will employ more programming tasks to further evaluate ADECK.

6.2 Threats to External Validity

ADECK is limited by the code sample sources of forums like Stack Overflow, in which millions of developers scan and search their desired information to learn API usages. If no Q&A pair in Stack overflow discusses a certain API, it would be impossible for ADECK to extract code samples with usage scenarios for it. In contrast, if there are more related Q&A pairs in Stack Overflow discussing an API, ADECK performs better on this API. Hence, ADECK may work better on more-common APIs, since there are likely to be more related Q&A pairs in Stack Overflow. Along with the rapid growth of Q&A pairs in Stack Overflow, ADECK has, without doubt, a great potential in obtaining high-quality code samples with their usage scenarios for more APIs.

We verify ADECK on JavaD and AndroidD without testing other API documentation, and the generalization of ADECK may be an external threat. JavaD is relatively mature and well-established, and has been applied to many areas. AndroidD is relatively new attracting a growing number of developers, and can be applied to mobile application development. Hence, they are representative to verify ADECK. The effectiveness of ADECK on other API documentation (e.g., C# API documentation in MSDN) is still unknown. In the future, we attempt to validate ADECK to enrich other API documentation with code samples and usage scenarios.

7 Related Work

7.1 API Documentation Analysis

There are a lot of research tasks focusing on API documentation. In this study, we pay attention to two closely related tasks, i.e., API documentation enrichment and content comprehension.

API documentation enrichment aims to augment API documentation with pieces of information. Treude *et al.* try to augment API documentation with insightful sentences in Stack Overflow [1]. Wu *et al.* propose *CoDocent* to trace relevant API documentation with fully-qualified APIs to help developers understand code [4]. Hoffman *et al.* enrich API documentation by executable test cases with expected output [45]. Stylos *et al.* develop a tool *Jadeite* to help developers discover and instantiate the correct APIs [46]. Subramanian *et al.* identify and link API elements in code snippets to API documentation [47]. In addition, Chen and Zhang propose a prototype to integrate crowdsourced FAQs into API documentation [11]. However, none of them enrich API documentation by code samples with usage scenarios, thus motivating us to propose ADECK.

Content comprehension aims to find important information in API documentation. Maalej *et al.* first analyze the knowledge patterns in API documentation [41]. Kumar *et al.* [48] automatically categorize the knowledge in API documentation based on the taxonomy in [41]. Monperrus *et al.* empirically study a specific category, namely directive, and propose a taxonomy containing 23 kinds of directives [49]. Zhong *et al.* try to detect errors in API documentation [50]. All the content comprehension studies only analyze the existing content in API documentation, whereas ADECK attempts to enrich the content of API documentation.

7.2 Mining Stack Overflow

Stack Overflow assembles crowd knowledge from millions of developers, and a lot of research tasks are proposed around it. Studies on mining Stack Overflow could be roughly divided into two categories, i.e., item prediction and knowledge utilization.

There are many items in the Q&A pairs in Stack Overflow, including user score, tag, favorite number, etc. It would be more convenient if these items can be predicted automatically. Lezina *et al.* build a classifier to predict whether a new question will be closed in the future [51]. Stanley *et al.* develop a Bayesian probabilistic model *ACT-R* to predict tags for new posts [52]. Choetkiertikul *et al.* predict the potential answerers for new questions [53].

The crowd knowledge in Stack Overflow can also be utilized by the other research tasks. Ponzanelli *et al.* use the knowledge in Stack Overflow to help developers comprehend and develop software [54]. Gao *et al.* fix recurring crash bugs by analyzing Q&A pairs [29]. Jiang *et al.* employ API related Q&A pairs as new features to discover relevant tutorial fragments [16].

Our study belongs to the category of knowledge utilization. Unlike the existing studies, ADECK tries to obtain high-quality code samples with usage scenarios from Stack Overflow and further embed them into API documentation.

7.3 API and Code Sample Recommendation

APIs play an important role in software development. Hence, there are a lot of studies in the literature focusing on mining and recommending APIs for developers. Gu *et al.* propose *DeepAPI*, a deep learning approach to generate API usage sequences for a natural language query [55]. Huang *et al.* propose *BIKER* that leverages Stack Overflow posts to obtain candidate APIs for a given programming task [56]. Xie *et al.* propose *MAPO* to generate API patterns by mining and ranking API sequence pattern clusters [57]. In addition, Wang *et al.* propose *UP-Miner* to further improve *MAPO* by mining succinct and high-coverage API usage patterns [58].

In addition to recommending APIs, researchers also try to recommend code samples for developers to help them resolve programming tasks. McMillan *et al.* design *Portfolio* to generate relevant C/C++ functions for a code search query [59], and Chan *et al.* further employ a graph search approach to improve *Portfolio* [60]. Liu *et al.* propose a web based tool *CodeNuance* to support code search with differencing and visualization [61]. In addition, Zhang *et al.* develop an automatic tool named *BDA* to recommend code samples mined from public software repositories and webpages [62].

These studies try to either recommend APIs or code samples for developers. In contrast, our study tries to enrich API documentation with code samples and usage scenarios to perform knowledge collaboration.

8 Conclusion and Future Work

Code samples with usage scenarios are extremely scarce in API documentation. It would be ideal if they can be enriched into API documentation automatically. Even though eXoaDocs is proposed to enrich API documentation, it cannot tackle the quality challenge and the mapping challenge. In this study, we propose a new approach named ADECK to enrich API documentation by code samples with usage scenarios from Stack Overflow. We show that the number of code-sample-illustrated API types in the ADECK-enriched API documentation is up to 5.76 times as many as that in the raw API documentation. In addition, the quality of code samples obtained by ADECK statistically outperforms the quality of code samples obtained by eXoaDocs. Developers using the ADECK-enriched documentation are 11.03% faster in terms of the average completion time and complete 14.29% more programming tasks than using the eXoaDocs-enriched API documentation.

For the future work, we have the following directions. First, we plan to introduce other API documentation to validate ADECK. Second, we try to consider the suggestions provided by the developers in the interview to further improve ADECK. Third, we intend to develop and release an automated tool encapsulating ADECK.

Acknowledgment

We thank the volunteers and developers for their contributions on the manual evaluation. This work is partially supported by the National Key Research and Development Plan of China under Grant No. 2018YFB1003900, the National Natural Science Foundation of China under Grant No. 61722202 and Grant No. 61602258, and the China Postdoctoral Science Foundation under Grant No. 2017M621247.

References

[1] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," *In Proceedings of the 38th International Conference on Software Engineering (ICSE 16)*, pp. 392–403, 2016.

[2] The Java SE API documentation of the ArrayList API: https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html, last verified date: 18/04/2019.

[3] H. Zhong and H. Mei, "An empirical study on api usages," *IEEE Transactions on Software Engineering*, p. to appear, 2018.

[4] Y. C. Wu, L. W. Mar, and H. C. Jiau, "Codocent: Support api usage with code example and api documentation," *In Proceedings of 5th International Conference on Software Engineering Advances (ICSEA 10)*, pp. 135–140, 2012.

[5] The Java SE API documentation version 7: http://docs.oracle.com/javase/7/docs/api/, last verified date: 18/04/2019.

[6] The Android API documentation version 4.4: https://developer.android.com/reference/packages.html, last verified date: 15/07/2018.

[7] J. X. Zhang, "What makes a good api tutorial?" *Technical Report*, 2016.

[8] G. Uddin and M. P. Robillard, "How api documentation fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, July 2015.

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

[10] J. Kim, S. Lee, S. Hwang, and S. Kim, "Enriching documents with examples: A corpus mining approach," *ACM Transactions on Information Systems (TOIS)*, vol. 1, p. 1, 2013.

[11] C. Chen and K. Zhang, "Who asked what: integrating crowd-sourced faqs into api documentation," *In Companion Proceedings of the 36th International Conference on Software Engineering*, pp. 456–459, 2014.

[12] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Transactions on Software Engineering*, p. to appear, 2018.

[13] S. Endrikat, S. Hanenberg, R. Robbes, and A. Stefik, "How do api documentation and static typing affect api usability?" *In Proceedings of the 36th International Conference on Software Engineering (ICSE 14)*, pp. 632–642, 2014.

[14] Y. H. Wu, S. W. Wang, C. P. Bezemer, and K. Inoue, "How do developers utilize source code from stack overflow?" *Empirical Software Engineering*, no. 2, pp. 1–37, 2018.

[15] C. Parnin, C. Treude, L. Grammel, and M. Storey, "Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow," *Technical Report*, 2012.

[16] H. Jiang, J. X. Zhang, X. C. Li, and et al., "A more accurate model for finding tutorial segments explaining apis," *In Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 16)*, pp. 157–167, 2016.

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

[18] H. Jiang, J. X. Zhang, Z. L. Ren, and T. Zhang, "An unsupervised approach for discovering relevant tutorial fragments for apis," *In Proceedings of the 39th International Conference on Software Engineering (ICSE 17)*, pp. 38–48, 2017.

[19] Z. Xing and E. Stroulia, "Api-evolution support with diff-catchup," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 818–836, 2007.

[20] G. Bavota, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "Api change and fault proneness: a threat to the success of android apps," *In Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering (ESEC/FSE 13)*, pp. 477–487, 2013.

[21] The Stack Overflow data dump files: https://archive.org/details/stackexchange, last verified date: 18/04/2019.

[22] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li, "Query expansion based on crowd knowledge for code search," *IEEE Transactions on Services Computing*, vol. 9, no. 5, pp. 771–783, 2016.

[23] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," *In Proceedings of International Conference on Mining Software Repositories (MSR 18)*, p. to appear, 2018.

[24] M. Ghafari, K. Rubinov, and M. Pourhashemk, "Mining unit test cases to synthesize api usage examples," *Journal of Software Evolution & Process*, p. e1841, 2017.

[25] L. de Souza, E. C. Campos, and M. Maia, "Ranking crowd knowledge to assist software development," *In Proceedings of the 22nd International Conference on Program Comprehension (ICPC 14)*, pp. 72–82, 2014.

[26] M. Linares-Vsquez, G. Bavota, M. Penta, R. Oliveto, and D. Poshyvanyk, "How do api changes trigger stack overflow discussions? a study on the android sdk," *In Proceedings of 22nd International Conference on Program Comprehension (ICPC 14)*, pp. 83–94, 2014.

[27] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example?: A study of programming q&a in stackoverflow," *In Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 12)*, pp. 25–34, 2012.

[28] A Q&A pair example in Stack Overflow: http://stackoverflow.com/questions/22658883/error-in-write-arraylist-string-to-file, last verified date: 18/04/2019.

[29] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei, "Fixing recurring crash bugs via analyzing q&a sites," *In Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 15)*, pp. 307–318, 2015.

[30] Y. Higo and S. Kusumoto, "How should we measure functional sameness from program source code? an exploratory study on java methods," *In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 14)*, pp. 294–305, 2014.

[31] The list of English stop words: http://xpo6.com/list-of-english-stop-words/, last verified date: 18/04/2019.

[32] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," *In Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC 08)*, pp. 172–181, June 2008.

[33] B. J. Frey and D. Dueck, "Clustering by passing messages between data points," *Science*, vol. 315, no. 5814, pp. 972–976, 2007.

[34] S. Sarkar, G. M. Rama, and A. C. Kak, "Api-based and information-theoretic metrics for measuring the quality of software modularization," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 14–32, 2007.

[35] S. Beyer, C. Macho, and M. Pinzger, "On android api classes and their references on stack overflow," *Technical Report*, 2016.

[36] D. S. Eisenberg, J. Stylos, and B. A. Myers, "Apatite:a new interface for exploring apis," *In Proceedings of the International Conference on Human Factors in Computing Systems*, pp. 1331–1334, 2010.

[37] The content of the survey on the developers: https://github.com/APIDocEnrich/ADECK, last verified date: 18/04/2019.

[38] X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, and B. Xu, "Revisit of automatic debugging via human focus-tracking analysis," *In Proceedings of the 38th International Conference on Software Engineering (ICSE 16)*, pp. 808–819, 2016.

[39] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," *In Proceedings of the ACM Sigsoft International Symposium on Foundations of Software Engineering (FSE 16)*, pp. 535–546, 2016.

[40] D. E. Johnson, "Crossover experiments," *WIREs Computational Statistics*, vol. 2, pp. 620–625, 2010.

[41] W. Maalej and M. P. Robillard, "Patterns of knowledge in api reference documentation," *IEEE Transactions on Software Engineering*, vol. 39, pp. 1264–1283, 2013.

[42] The official website of the Camtasia Studio: https://camtasia-studio.en.softonic.com/, last verified date: 18/04/2019.

[43] J. Coelho and M. T. Valente, "Why modern open source projects fail," *In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE 17)*, pp. 186–196, 2017.

[44] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," *In Proceedings of the ACM Sigsoft International Symposium on Foundations of Software Engineering (FSE 16)*, pp. 858–870, 2016.

[45] D. Hoffman and P. Strooper, "Api documentation with executable examples," vol. 66, no. 2, pp. 143–156, May 2003.

[46] J. Stylos, A. Faulring, Z. Yang, and B. A. Myers, "Improving api documentation using api usage information," *In Proceedings of 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 119–126, 2009.

[47] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," *In Proceedings of the 36th International Conference on Software Engineering (ICSE 14)*, pp. 643–652, 2014.

[48] N. Kumar and P. Devanbu, "Ontocat: Automatically categorizing knowledge in api documentation," 2016.

[49] M. Monperrus, M. Eichberg, E. Tekes, and et al., "What should developers be aware of? an empirical study on the directives of api documentation," *Empirical Software Engineering*, vol. 17, pp. 703–737, 2012.

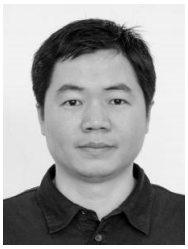
[50] H. Zhong and Z. D. Su, "Detecting api documentation errors," *In Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications (OOPSLA 13)*, pp. 803–816, 2013.

[51] G. E. Lezina and A. M. Kuznetsov, "Predict closed questions on stackoverflow," *In Proceedings of the 9th Spring Researchers Colloquium on Database and Information Systems*, pp. 10–14, 2013.

[52] C. Stanley and M. D. Byrne, "Predicting tags for stack overflow posts," *In Proceedings of ICCM*, 2013.

[53] M. Choetkiertikul, D. Avery, H. K. Dam, T. Tran, and A. Ghose, "Who will answer my question on stack overflow?" *In Proceedings of the 24th Australasian Software Engineering Conference*, pp. 155–164, 2015.
[54] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Leveraging crowd knowledge for software comprehension and development," *In Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 13)*, pp. 57–66, 2013.
[55] X. D. Gu, H. Y. Zhang, D. M. Zhang, and S. Kim, "Deep api learning," *In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 16)*, pp. 631–642, 2016.
[56] Q. Huang, X. Xia, Z. C. Xing, D. Lo, and X. Y. Wang, "Api method recommendation without worrying about the task-api knowledge gap," *In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 18)*, pp. 293–304, 2018.
[57] 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, 2006.
[58] J. Wang, Y. N. Dang, H. Y. Zhang, K. Chen, T. Xie, and D. M. Zhang, "Mining succinct and high-coverage api usage patterns from source code," *In Proceedings of the International Workshop on Mining Software Repositories (MSR 13)*, pp. 319–328, 2013.
[59] C. Mcmillan, M. Grechanik, D. P. Denys, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," *In Proceedings of the International Conference on Software Engineering (ICSE 11)*, pp. 111–120, 2011.
[60] W. Chan, H. Cheng, and D. Lo, "Searching connected api subgraph via text phrases," *In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE 12)*, p. NO. 10, 2012.
[61] W. Liu, X. Peng, Z. Xing, J. Li, B. Xie, and W. Zhao, "Supporting exploratory code search with differencing and visualization," *In Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER 18)*, pp. 300–310, 2018.
[62] H. Y. Zhang, A. Jain, G. Khandelwal, C. Kaushik, S. Ge, and W. X. Hu, "Bing developer assistant: improving developer productivity by recommending sample code," *In Proceedings of the ACM Sigsoft International Symposium on Foundations of Software Engineering (FSE 16)*, pp. 956–961, 2016.

Jingxuan Zhang is a lecturer of College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China. Zhang received the Ph.D. degree in software engineering from the Dalian University of Technology, China. His current research interests include mining software repositories and software data analytics.



He Jiang is an awardee of the NSFC Excellent Young Scholars Program in 2017. He is currently a professor with Dalian University of Technology and an adjunct professor with Beijing Institute of Technology. His current research interests include search-based software engineering and mining software repositories. He has published over 60 referred papers on journals and international conferences, including IEEE Trans. Software Engineering, IEEE Trans. Knowledge and Data Engineering, ICSE, SANER, etc., supported by the Program for New Century Excellent Talents in University and the National Science Fund for Excellent Young Scholars. In addition, he serves as the guest editors of some journals and magazines, including IEEE Computational Intelligence, Journal of Computer Science and Technology, Frontiers of Computer Science, etc.

Zhilei Ren is an associate professor of School of Software, Dalian University of Technology, China. Ren received the B.Sc. degree in software engineering and the Ph.D. degree in computational mathematics from the Dalian University of Technology, Dalian, China, in 2007 and 2013, respectively. His current research interests include evolutionary computation and its applications in software engineering.

Tao Zhang is an associate professor of College of Computer Science and Technology, Harbin Engineering University, China. Zhang received the Ph.D. degree in computer science from the University of Seoul, South Korea in 2013. He was a postdoctoral fellow at Department of Computing, Hong Kong Polytechnic University from Nov. 2014 to Nov. 2015. His research interests include data mining and software maintenance.

Zhiqiu Huang is a professor of College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China. He received the Ph.D. degree in computer science from the Nanjing University of Aeronautics and Astronautics in 1999. He has authored over 80 papers in referred journals and proceedings in the areas of software engineering and knowledge engineering. His research interests include software engineering, formal methods, and knowledge engineering.