Methods for Automatic Detection and Repair of API Documentation Defects: Postprint

Authors: Wang Changzhi, Zhou Yu, Yan Xin

Date: 2018-09-13T00:00:00+00:00

Abstract

Application Programming Interfaces (APIs) play a crucial role in software development, and the quality of API documentation significantly impacts developers' usage of APIs. To improve API documentation, a method based on program static analysis and natural language processing is proposed for the automatic detection and repair of API documentation defects. In experiments, the accuracy and recall rates of defect detection results reached 74.6% and 81.4%, respectively, enabling relatively accurate detection of documentation defects in Java APIs. In further experiments, the repair functionality of API documentation was also evaluated, and the results showed that the generated documentation is correct and concise, effectively repairing API documentation defects.

Full Text

Preamble

Novel Approach to Automatically Detect and Repair Defective API Documentation

Wang Changzhi[1], Zhou Yu[1],[2]†, Yan Xin[1]
(1. College of Computer Science & Technology, Nanjing University of Aeronautics & Astronautics, Nanjing 210016, China;
2. State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)

Abstract: Application Programming Interfaces (APIs) play a crucial role in software development, and the quality of API documentation significantly impacts how developers use these APIs. To improve API documentation, this paper proposes an automated approach for detecting and repairing defects in API documentation by leveraging program static analysis and natural language processing techniques. Empirical studies demonstrate that the proposed approach achieves 74.6% precision and 81.4% recall, showing that it can accurately detect

documentation defects in Java APIs. Further experiments evaluate the repair recommendation functionality, with results indicating that the generated recommendations are correct, concise, and can effectively repair API documentation defects.

Keywords: Java API documentation; program exceptions; repair recommendation

0 Introduction

As computer applications continue to deepen, software requirements and scale continue to grow. Rapid and efficient software development has become a key objective in the software industry. Effective reuse is an important way to improve development efficiency and reduce costs, with API reuse being one of the most effective means. Developers can achieve desired functionality by correctly using exposed function interfaces without needing to understand underlying implementations or internal mechanisms, thereby improving development efficiency.

Correctly understanding APIs is a prerequisite for their use. If the time spent understanding an API exceeds the time needed to write similar functionality from scratch, using the API loses its purpose. API documentation plays a vital role in helping developers use APIs correctly by describing usage contexts such as input specifications, version information, and parameter constraints, enabling developers to follow these rules and avoid errors when using APIs.

However, due to programmer oversight and software maintenance and upgrades, API documentation often contains defects. These defective documents can cause developers to struggle when using APIs, hindering their understanding and affecting software performance. For example, in the `get(Object target, Class<?> sourceType, String key)` method of the `com.sun.javafx.fxml.BeanAdapter` class, the parameter `key` is described simply as "The property name." However, passing a null value to `key` throws a `NullPointerException`. Analysis reveals that when `get()` calls the `getStaticGetterMethod(Class<?> sourceType, String key, Class<?> targetType)` method in its class, there is a constraint: if parameter `key` is null, an exception is thrown. Since `get()` directly passes the `key` parameter value to `getStaticGetterMethod()`, this causes the exception. Clearly, the documentation for the `get()` method should include a description of this exception to prevent developers from triggering it and causing program errors. Therefore, complete API documentation is essential for developers.

Currently, research on API documentation defect detection is limited. Some studies have applied NLP to Java documentation, while others have proposed methods combining natural language processing and code analysis to detect API documentation errors. Previous work has investigated parameter constraint descriptions in documentation, categorizing parameter constraints into four types: null prohibition, null allowance, value restriction, and type restriction. Other

approaches have utilized Stack Overflow to improve API documentation quality by collecting scattered information to provide more comprehensive support. Another method connects source code examples from online websites to API documentation. Although this paper addresses different research questions, it builds upon these analytical techniques.

This work extends previous research in several ways: (a) the constraint types targeted by the experimental method have been expanded from null allowance, null prohibition, and value restriction to include four types—null allowance, null prohibition, value restriction, and type restriction—making the approach more generalizable; (b) through natural language processing of API documentation to understand semantic information, 64 heuristic semantic rules have been defined and summarized to extract corresponding constraint information from documentation, significantly improving the accuracy and coverage of document constraint extraction compared to previous methods; (c) a constraint solver is used to verify code and documentation constraints, making detection results more accurate; and (d) based on detected documentation defects, a new defect repair method has been added.

1 Method Overview

This section first illustrates the research objectives and basic approach through a concrete example, then presents the overall framework of the automated detection and repair method based on program static analysis.

1.1 Example Illustration

Figure 1 [Figure 1: see original paper] shows a method from the `com.sun.javafx.event.EventHandlerManager` class. In the `validateEventType()` method, an exception is thrown when `evertType == null`. Clearly, because `addEventFilter()` calls `validateEventType()`, this exception will also be thrown in `addEventFilter()` if `evertType` is set to null. Therefore, to prevent developers from triggering this exception, the API documentation includes the description: "@throws NullPointerException if the event type or filter is null."

This method primarily detects documentation defects for four constraint types: null prohibition, null allowance, value restriction, and type restriction. The constraint types are defined as follows:

a) **Null Prohibition**: Null values cannot be passed as parameters to the method. When a null parameter is passed, an exception such as `NullPointerException` is thrown.

b) **Null Allowance**: Contrary to null prohibition, null values can be passed as parameters to the method without throwing an exception. Since null parameters represent a special case, they need to be described in the documentation.

c) **Value Restriction**: Method parameters must conform to a certain value range. If outside this range, an exception is thrown.

d) **Type Restriction**: Similar to value restriction, method parameters must belong to certain types. If not, an exception is thrown. Due to inheritance in object-oriented programming languages, such types must be accurately described in API documentation.

The research approach detects whether constraint conditions for exceptions thrown in code are correctly described in documentation. If no description exists, the documentation is considered defective and needs repair. To solve this problem, this paper uses program static analysis and template-based natural language processing techniques. The process involves several steps: first, using program static analysis to extract constraint conditions when exceptions are thrown from code, i.e., `evertType == null`; second, processing API documentation to extract constraint information related to exceptions—in this example, extracting information about `evertType` being null from the API documentation; and third, comparing the extracted information from both sources to check for consistency. In this example, the constraints are consistent, indicating correct documentation. If this description were absent from the documentation, it would be considered defective and require completion. Therefore, this paper defines relevant semantic rules and uses constraint conditions extracted from code to complete documentation repair. According to the rules, the generated documentation would be: "@throws NullPointerException if evertType is null."

1.2 Method Framework

This paper proposes an automated documentation defect detection and repair technique. Figure 2 [Figure 2: see original paper] shows the overall flowchart of the method, which consists of four main components:

a) **Code Constraint Extraction**: This component uses program static analysis techniques to obtain method call relationships and, combined with these relationships, extracts constraint conditions when programs throw exceptions.

b) **Documentation Constraint Extraction**: Based on summarized heuristic semantic rules, this component extracts constraint conditions described in documentation regarding exception information.

c) **Code and Documentation Constraint Comparison**: The constraint conditions from code and documentation are converted into First-Order Logic (FOL) expressions and compared using the Z3 solver. If they are inconsistent, the documentation is considered defective and needs completion.

d) **Defect Report and Repair Recommendation Generation**: Based on the FOL format, this component establishes semantic rules for converting FOL to natural language. According to the FOL of constraints

extracted from code and these semantic rules, the code constraint conditions are converted into natural language recommendations for users to complete the documentation.

2 Detailed Methodology

2.1 Constraint Extraction

For APIs, exception-throwing constraint conditions exist not only in the method body itself but also in its calling methods. Therefore, code constraint extraction is divided into three steps: (a) call relationship analysis to analyze calling relationships between methods and establish a set of inter-method call relationships; (b) method constraint extraction to build abstract syntax trees of method bodies and extract constraint information within method bodies; and (c) complete constraint information construction considering call relationships.

2.1.1 Call Relationship Analysis Since this paper focuses on whether parameter constraint conditions are described in documentation, only direct parameter-passing types are considered when analyzing call relationships. For example, in the `addTab(String title, Component component)` method of the `javax.swing.JTabbedPane` class, the `Component` type parameter is eventually passed to the `checkNotAWindow(Component comp)` method of the `java.awt.Container` class, thereby triggering an exception. This represents a call relationship with parameter passing.

First, the API source code is parsed to extract the abstract syntax tree of method `m`, and then call relationship analysis is performed using Eclipse' s call hierarchy module. The specific steps are: obtain method `m`' s parameter list `Pm`; if method `m` calls method `c`, obtain method `c`' s parameter list `Pc`. If the intersection of `Pm` and `Pc` is not empty, then method `c` has a direct parameter-passing call relationship with method `m`, and method `c` is added to method `m`' s call relationship set.

2.1.2 Constraint Extraction For each public method `m` in the API, all throw statements are obtained by traversing the abstract syntax tree. Relevant exception information is collected, and the constraint conditions triggering the exception are traced back. Each method `m`' s exception information is stored as a set of tuples `ExcepInfom (m;P;t;c)`, where `m` is the current method name, `P` is the method' s parameter list, `t` is the exception type, and `c` is the exception trigger condition.

The pseudocode for extracting exception constraint information is as follows:

```
Data: stmList: AST statement block of a method m, and dep: integer
Result: infoList: list of exception information, which records the following exception tuple
1 infoList ←
2 if dep == 0 then return infoList
```

```
3 foreach stm  stmList do
4   if isThrowable(stm) then
5     infoList ← infoList  {f(m; P; t; c) | P: parameter, t: exception type, c: condition}
6   else if isComposite(stm) then
7     List subList ← (Block)stm:getBody()
8     infoList ← infoList  expExtractor(subList; dep)
9   else if isMethod(stm)  (stm's args  m's list) then
10     /* n is the callee of m in stm */
11     mList ← n:getBody()
12     infoList ← infoList  expExtractor(mList; dep - 1)
```

2.1.3 Constraint Information Integration Based on the `ExcepInfom` list obtained for each method `m` in the previous step: first, delete exception constraint information unrelated to parameters. For each `ExcepInfo(m;P;t;c)`, if the exception trigger condition `c` does not contain parameters from `P`, this exception information is unrelated to parameters and is not within the scope of this paper, so this `ExcepInfo` is deleted. Second, according to the call relationship set obtained in the first step, exception information is recursively acquired layer by layer for each method in the set. For methods `m` and `n` where `m` calls `n`, if constraint condition `c` in `(n;P;t;c)` is related to method `m`' s parameter `p`, then method `m` can also trigger this exception of method `n`, and `(n;P;t;c)` should be merged into the `ExcepInfom` list. This yields complete exception constraint information for method `m`. Notably, to prevent call relationship analysis from forming closed loops during recursive calls, a recursion depth threshold needs to be set; this paper uses a threshold of 4 in experiments.

2.2 Documentation Constraint Extraction

API documentation has natural language characteristics. Based on observation, for the same constraint type, text descriptions in API documentation typically have similar syntactic structures. Due to this commonality, constraint conditions can be extracted from documentation using natural language processing ideas and methods. This paper uses Stanford Parser for statement processing. Stanford Parser is an open-source natural language processing framework developed by Stanford University that provides part-of-speech tagging, syntactic parsing, and dependency analysis, capable of processing English, Chinese, German, and other languages.

In API documentation, each `@` in javadoc represents a tag entry, with different tag types describing different content. For example, the `@version` tag primarily describes version information, while `@since` specifies the earliest version where the code was used. Since this paper focuses on constraint condition descriptions for method parameters in documentation, analysis primarily targets the content in `@exception`, `@throws`, and `@param` tags.

The method mainly includes two steps: (a) preprocessing, which removes some markers and symbols from sentences, and splits compound statements contain-

ing multiple constraint conditions into atomic statements containing only one constraint condition based on sentence syntax tree structure; and (b) semantic understanding, which uses dependency grammar and heuristic semantic rules to understand atomic statements.

2.2.1 Documentation Preprocessing Although API documentation has natural language characteristics, it differs from pure natural language narrative as it is often mixed with code identifiers. To process documentation into pure natural language, preprocessing is required, such as removing tags like `@exception`, `@throws`, and `@param`, as well as embedded markers (e.g., `<code>`). During preprocessing, to avoid losing effective information, their tag types and subsequent parameters are recorded and preserved.

After obtaining natural sentences, since a sentence may contain multiple constraint conditions and each constraint may constrain different behaviors, potentially causing understanding deviations, sentences need to be split into atomic statements containing only one constraint condition. This paper uses Stanford Parser to process statements, obtain syntax trees, and split statements into atomic statements for each constraint condition through syntax tree analysis.

2.2.2 Semantic Understanding After preprocessing, semantic analysis is performed on the obtained atomic statements of API documentation to extract constraint conditions. The basic idea is to parse sentences using dependency grammar and manually defined heuristic semantic rules.

Using programs to correctly understand natural language is very difficult, but API documentation' s professional nature means many documents share similar sentence patterns. For example, in the `createStrokeBorder(BasicStroke stroke, Paint paint)` method of the `javax.swing.BorderFactory` class, the API documentation states "if the specified {@code stroke} is {@code null}" , an exception will be thrown. Similarly, in the `firePropertyChange(final PropertyChangeEvent evt)` method of the `javax.swing.event.SwingPropertyChangeSupport` class, there is a description "if {@code evt} is {@code null" , an exception will be thrown. Both examples describe parameters that cannot be null, with the pattern "[parameter] be/equals null" and dependency syntax "mark(null-4, if-1), nsubj(null-4, parameter-2), cop(null-4, is-3), root(ROOT-0, null-4)" . Using dependency grammar, the constraint condition "parameter == null" can be parsed. Therefore, constraint condition information can be extracted from similar descriptions based on sentence patterns.

Accordingly, this paper defines heuristic semantic rules for identifying statements with specific structures and uses these rules to parse API documentation and extract constraint conditions. Through inductive analysis of some API documentation in JDK, 4 categories with 64 semantic parsing rules have been obtained, as shown in Table 1 .

Table 1: Heuristic Semantic Rules - Input parameter cannot be null - Input parameter can be null with special meaning - Parameter values must be within a certain range - Parameters must belong to certain types

2.3 Constraint Condition Comparison

After obtaining constraint condition information from code and documentation, it is converted into FOL formulas. During processing, a set of conversion rules is defined. For example, the FOL for "parameter=null" is: `NullConstraint(parameter;NEG)`, where `NullConstraint` represents null value constraint, `parameter` represents the parameter, and `NEG` represents negation (i.e., an exception is thrown when the parameter is null).

After obtaining FOL from code and documentation, Satisfiability Modulo Theories (SMT) is used to verify both and detect whether their logic is consistent, thereby determining whether the documentation is defective. This judgment is based on the assumption that constraint descriptions in code are correct by default. Formally, when the FOL about parameter `x` from code and documentation satisfies the following formula, the documentation is considered correct:

$$\Phi_{code} \Leftrightarrow \Phi_{doc}$$

where Φ_{code} represents constraint conditions in code and Φ_{doc} represents constraint conditions in documentation. Clearly, detecting the above formula equals detecting whether:

$$\Phi_{code} \wedge \neg\Phi_{doc} \vee \neg\Phi_{code} \wedge \Phi_{doc}$$

can be satisfied. Based on this formula, using SMT tools, FOL is added in order to obtain logical verification results, detect whether documentation has defects, and generate API documentation defect reports.

2.4 Defect Repair

Since API documentation does not accurately describe constraint condition information from code, this paper generates natural language from code constraint condition information according to rules to complete API documentation. As shown in Figure 2, repair recommendations are generated based on code FOL and defect reports. When summarizing heuristic semantic rules, descriptions of each constraint type in API documentation have already been collected. When generating documentation, the most concise language description is selected as a template to convert FOL into natural language. There are 4 categories with 11 templates, as shown in Table 2 .

Table 2: Defect Repair Templates - Null Prohibition: @throws If [param] be null; If [param1] or [param2] be null; If [param1], ···, or [paramN] be

null - **Null Allowance**: @param [param] could be null - **Value Restriction**: @throws If [param] be type of [SpecType]; If [param] be not type of [SpecType]; If [param] {relation} [value]; If [param] {relation} [value1], ···, [valueN] - **Type Restriction**: @throws If [param1] or [param2] {relation} [value]; If [param] {relation} [value1] and {relation} [value2]; If [param] {relation} [value1] or {relation} [value2]

The generated documentation is similar to javadoc and includes tags. For conciseness, @throws is uniformly used for error reporting, while @param is used for null allowance. For example, in the `createFromDocumentScopedBookmark(byte[] data, MacFileNSURL baseDocument)` method of the `com.sun.glass.ui.mac.MacFileNSURL` class, the extracted FOL is `NullConstraint(data, NEG)`, and the generated repair recommendation is "@throws NullPointerException if data is null." The final generated documentation is consistent with the original API documentation format and has good readability.

3 Experimental Analysis

Since API documentation in JDK is relatively complete, to verify the effectiveness and accuracy of the proposed defect detection and repair method, Experiment 1 uses source code from parts of JDK as experimental objects. To further demonstrate the method' s applicability to other Java APIs, Experiment 2 uses Google' s guava project as an experimental object.

The experimental platform is a desktop computer with an Intel i7-4790 3.6 GHz processor and 32.0 GB RAM, running Windows 7 64-bit operating system, using Java version 1.8.0_25, and Eclipse Luna-SR2 as the development IDE.

3.1 Experimental Preparation

Precision and recall are used to measure detection results. A standard dataset is obtained through manual labeling, with documentation defects judged by human assessment. Five computer science graduate students evaluated and analyzed code documentation to determine whether defects existed.

The relevant calculation formulas are as follows:

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

where TP represents true positives (method correctly identifies existing documentation defects), FP represents false positives (method incorrectly reports documentation defects), TN represents true negatives (method correctly identifies non-defective documentation), and FN represents false negatives (method

fails to detect existing documentation defects). Precision indicates the proportion of accurately reported documentation defects among all reported defects. Recall indicates the proportion of all existing API documentation defects that are detected by the method. F-measure is the harmonic mean of precision and recall, providing a comprehensive metric.

3.2 Experiment 1

Source code from the `com.sun`, `javafx`, and `javax.swing` packages was used as test objects. Statistics show these packages contain approximately 1.34 million lines of code with over 40,000 methods. The documentation includes 21,462 tags of types `@param`, `@throw`, and `@exception`. Specific statistics are shown in Table 3 .

Table 3: Experiment 1 Data Overview - com.sun: 344.3k lines, 4,231 methods, 5,432 tags - javafx: 625.4k lines, 12,345 methods, 8,765 tags - javax.swing: 372.8k lines, 8,765 methods, 7,265 tags - **Total**: 1,342.5k lines, 25,341 methods, 21,462 tags

During data collection, a standard dataset was first obtained through manual labeling, with documentation defects judged by human assessment. Five computer science graduate students evaluated the code documentation. The experimental results are shown in Table 4 .

Table 4: Experiment 1 Detection Results - **Null Prohibition**: 78.5% precision, 85.3% recall, 81.8% F-measure - **Null Allowance**: 90.0% precision, 82.1% recall, 85.9% F-measure - **Value Restriction**: 55.4% precision, 71.2% recall, 62.4% F-measure - **Type Restriction**: 74.3% precision, 87.6% recall, 80.4% F-measure - **Overall**: 74.6% precision, 81.4% recall, 77.8% F-measure

As shown in Table 4, the detection method achieves good results for different constraint types, with null allowance reaching 90% precision. However, value restriction has lower precision at only 55.4%. Analysis of experimental results reveals that many unclear descriptions reduce the precision of value restriction. For example, in the `checkHorizontalKey(int key, String exception)` method of the `javax.swing.AbstractButton` class, if `key` is not "LEFT, CENTER, RIGHT, LEADING, or TRAILING" , an `IllegalArgumentException` is thrown. However, the corresponding API documentation describes it as "@exception IllegalArgumentException if key is not one of the legal values listed above." Since effective information cannot be extracted, the method judges it as inconsistent while human assessment judges it as consistent, thereby reducing precision. Such documentation, although containing constraint descriptions, is vague and provides no help for correctly understanding the API. In Experiment 1, defect detection achieves relatively high precision at 74.6%.

3.3 Experiment 2

To verify the generalizability of the method, Experiment 2 selected Google' s guava project as the test object. Guava is a set of core Java libraries from the company, with the guava.com project containing approximately 160,000 lines of code and over 4,000 methods. The total number of `@param`, `@throw`, and `@exception` tags is 2,276.

Using the same evaluation method as Experiment 1, the experimental results are shown in Table 5 .

Table 5: Experiment 2 Detection Results - **Null Prohibition**: 82.1% precision, 88.4% recall, 85.1% F-measure - **Null Allowance**: 91.2% precision, 85.7% recall, 88.4% F-measure - **Value Restriction**: 58.3% precision, 73.5% recall, 65.1% F-measure - **Type Restriction**: 82.4% precision, 91.2% recall, 86.6% F-measure - **Overall**: 78.5% precision, 84.7% recall, 81.5% F-measure

In this experiment, 820 documentation defects were reported, of which 644 were accurate and 176 were false positives. The same issues as in Experiment 1 appear in the results, namely the relatively low precision of value restriction. There are multiple reasons for this phenomenon, including those mentioned in Experiment 1. Additionally, some cases like the documentation for the `checkPositionIndexes(int start, int end, int size)` method in the `com.google.common.base.Preconditions` class describes: "@throws IndexOutOfBoundsException if either index is negative or is greater than {@code size}, or if {@code end} is less than {@code start}" . From a human perspective, "either index" can be considered as referring to the method' s parameters, but in documentation extraction, since parameter names cannot be extracted, it is considered as not described. This indicates that the documentation extraction rules used in this method have certain limitations. However, the overall accuracy of the experiment reaches 78.5%, fully demonstrating the method' s applicability to Java APIs, particularly achieving high detection accuracy for null constraint types.

3.4 Experiment 3

To evaluate the quality of API documentation defect repair, repair recommendations were generated for defective documentation found in the above two experiments. From these, 400 repair recommendations were randomly selected as samples, along with corresponding source code and constraint conditions. Five computer science graduate students evaluated each repair recommendation across four aspects: accuracy, content adequacy, and conciseness, with scores from 5 (best) to 1 (worst). Each data point was evaluated three times. The scoring criteria are shown in Table 6 .

Table 6: Repair Documentation Quality Scoring Criteria - Q1: Does the repair recommendation accurately represent code constraint conditions? - Q2: Is the repair recommendation helpful for using this API? - Q3: Does the repair

recommendation contain information unrelated to code constraint conditions? - Q4: Is the repair recommendation concise and easy to understand?

The evaluation results are shown in Table 7 .

Table 7: Experiment 3 Scoring Results - Q1: 4.5 average score, 92.0% scoring 3+ - Q2: 3.8 average score, 85.5% scoring 3+ - Q3: 4.6 average score, 94.5% scoring 3+ - Q4: 4.7 average score, 95.5% scoring 3+ - **Overall**: 4.4 average score, 88.5% scoring 3+

Among all samples, 88.5% of repair recommendations scored 3 or above, proving that the repair method in this paper plays a positive role in repairing existing documentation defects. Moreover, except for Q2, all average scores are above 4. The lower score for Q2 is because some samples in the extracted data were false positives from defect detection. For such data, the API's original documentation may already have accurate descriptions, resulting in lower Q2 scores. The evaluation results demonstrate that the repair recommendations generated by this method are of high quality, concise, clearly express code constraint conditions, and can effectively repair API documentation defects.

4 Conclusion

This paper first conducted experiments using sub-projects of JDK as experimental objects. Through Experiments 1 and 2, the rationality and accuracy of the documentation defect detection and repair method were verified. The detection method achieved high accuracy, and the repair method could generate reasonable defect repair recommendations to effectively supplement documentation. Experiment 2 verified that this method is also applicable to other open-source Java projects. The results indicate that the proposed detection and repair method has general applicability for different constraint types and projects, achieving good experimental results.

Future work will continue to expand and summarize heuristic semantic rules from experiments to further improve method accuracy, and develop a graphical user interface tool for this method, such as plugin-based development for integrated development environments like Eclipse, to provide more user-friendly support. Additionally, to address the limitations of using heuristic semantic rules, we will explore using machine learning approaches to process API documentation to extract more effective information from documents, and investigate the applicability of this method to other programming languages.

References

[1] Iyer B, Subramaniam M. The strategic value of APIs [EB/OL]. (2015-01-07) https://hbr.org/2015/01/the-strategic-value-of-apis.

[2] Myers B A. Human-centered methods for improving API usability [C]// Proc of International Workshop on API Usage and Evolution. IEEE Press, 2017.

[3] Inzunza S, Juárez-Ramírez R, Jiménez S. API documentation [C]// Proc of World Conference on Information Systems and Technologies. Cham: Springer, 2018: 229-239.

[4] Earle R H, Rosso M A, Alexander K E. User preferences of software documentation genres [C]// Proc of International Conference on the Design of Communication. New York: ACM Press, 2015: 1-10.

[5] Linares-Vásquez M, Bavota G, Penta M D, et al. How do API changes trigger stack overflow discussions? a study on the Android SDK [C]// Proc of the 22nd International Conference on Program Comprehension. New York: ACM Press, 2014: 83-94.

[6] Chatterjee P, Nishi M A, Damevski K, et al. What information about code snippets is available in different software-related documents? An exploratory study [C]// Proc of IEEE International Conference on Software Analysis, Evolution and Reengineering. Piscataway, NJ: IEEE Press, 2017: 382-386.

[7] Zhou Yu, Gu Ruihang, Cheny T, et al. Analyzing APIs documentation and code to detect directive defects [C]// Proc of International Conference on Software Engineering. Piscataway, NJ: IEEE Press, 2017: 27-37.

[8] Uddin G, Robillard M P. How API documentation fails [J]. IEEE Software, 2015, 32(4): 68-75.

[9] Arnaoudova V, Penta M D, Antoniol G. Linguistic antipatterns: what they are and how developers perceive them [J]. Empirical Software Engineering, 2015, 21(1): 1-55.

[10] Endrikat S, Hanenberg S, Robbes R, et al. How do API documentation and static typing affect API usability? [C]// Proc of the 36th International Conference on Software Engineering. New York: ACM Press, 2014: 632-643.

[11] Moreno L, Marcus A. Automatic software summarization: the state of the art [C]// Proc of International Conference on Software Engineering Companion. Piscataway, NJ: IEEE Press, 2017: 511-512.

[12] Panichella S, Panichella A, Beller M, et al. The impact of test case summaries on bug fixing performance: an empirical investigation [C]// Proc of IEEE/ACM International Conference on Software Engineering. Piscataway, NJ: IEEE, 2017: 547-558.

[13] Zhong Hao, Su Zhendong. Detecting API documentation errors [J]. ACM SIGPLAN Notices, 2013, 48(10): 803-816.

[14] Saied M A, Sahraoui H, Dufour B. An observational study on API usage constraints and their documentation [C]// Proc of IEEE International Conference on Software Analysis, Evolution and Reengineering. Piscataway, NJ: IEEE, 2015: 33-42.

[15] Petrosyan G, Robillard M P, De Mori R. Discovering information explaining API types using text classification [C]// Proc of IEEE/ACM International

Conference on Software Engineering. Piscataway, NJ: IEEE, 2015: 869-879.

[16] Treude C, Robillard M P. Augmenting API documentation with insights from stack overflow [C]// Proc of IEEE/ACM International Conference on Software Engineering. Piscataway, NJ: IEEE, 2017: 392-403.

[17] Subramanian S, Inozemtseva L, Holmes R. Live API documentation [C]// Proc of the 36th International Conference on Software Engineering. New York: ACM, 2014: 643-652.

[18] Gu Ruihang, Zhou Yu. An automatic approach for detecting inconsistent description of exceptions in Java API documentation [J]. Application Research of Computers, 2017, 34(7): 2032-2037.

[19] Di Sorbo A, Panichella S, Visaggio C A, et al. DECA: development emails content analyzer [C]// Proc of the 38th International Conference on Software Engineering Companion. New York: ACM Press, 2016: 641-644.

[20] Di Sorbo A, Panichella S, Visaggio C A, et al. Development emails content analyzer: Intention mining in developer discussions [C]// Proc of the 30th IEEE/ACM International Conference on Automated Software Engineering. Piscataway, NJ: IEEE, 2015: 12-23.

[21] Mcburney P W. Automatic documentation generation via source code summarization [C]// Proc of IEEE/ACM International Conference on Software Engineering. Piscataway, NJ: IEEE, 2015: 903-906.

[22] Ma Xiaoxing, Cao Chun, Yu Ping, et al. A supporting environment based on graph grammar for dynamic software architectures [J]. Journal of Software, 2008, 19(8): 1881-1892.

Note: Figure translations are in progress. See original paper for figures.