Delfim *et al. Journal of the Brazilian Computer Society* (2016) 22:9
DOI 10.1186/s13173-016-0049-0

Journal of the Brazilian Computer Society

RESEARCH **Open Access**

Redocumenting APIs with crowd knowledge: a coverage analysis based on question types

Fernanda Madeiral Delfim[1*], Klérisson V. R. Paixão[1], Damien Cassou[2] and Marcelo de Almeida Maia[1]

Abstract

Background: Software libraries and frameworks play an important role in software system development. The appropriate usage of their functionalities/components through their APIs, however, is a challenge for developers. Usually, API documentation, when it exists, is insufficient to assist them in their programming tasks. There are few API documentation writers for the many potential readers, resulting in the lack of explanations and examples concerning different scenarios and perspectives. The interaction of developers on the Web, on the other hand, generates content concerning APIs from different perspectives, which can be used to document APIs, also known as crowd documentation.

Methods: In this paper, we present a study regarding the knowledge generated by the crowd on the Stack Overflow question-and-answer website. Our main goal is to understand how the crowd can contribute for API documentation on two programming tasks: how to implement a scenario using an API (*how-to-do-it*), and how to fix domain-independent bugs in an existing code where there was a misunderstanding regarding the usage of an API (*debug-corrective*). We classified questions available on Stack Overflow by the main concerns of askers, and we used those classified as *how-to-do-it* and *debug-corrective* to analyze the coverage of API elements on the discussions related to such questions. Our cases included the well-known and popular Swing and Android APIs.

Results: Our main findings showed that the crowd provides more content for *debug-corrective* tasks than for *how-to-do-it* tasks, regardless of the API. Android API elements are more discussed by the crowd compared to Swing. Moreover, we observed that some API elements are frequently mentioned together in discussions, and that there is a strong association between API coverage on Stack Overflow and its usage in real software systems.

Conclusions: Crowd documentation may not be a complete substitute for official documentation because of its partial coverage, especially for *how-to-do-it* tasks. However, it can still significantly enhance the existent documentation, especially for the most commonly used API elements, providing code samples and explanations on a large variety of usage nuances. Finally, taking advantage of the high coverage for *debug-corrective* tasks, a new kind of debugging assistant may be conceived.

Keywords: API documentation, Crowd knowledge, Stack Overflow, Question classification, Coverage analysis

Introduction

New development platforms are being deployed at an unprecedented pace. Software developers are required to deeply learn their respective APIs (application programming interface) to take maximum advantage of the underlying innovations, while avoiding misuses that could decrease the final product quality. Meanwhile, developers have reported that inadequate or absent resources for learning APIs, e.g., documentation, is a major obstacle for adequate learning [1, 2].

The social interaction of developers in blogs, forums, and question-and-answer (Q&A) websites generates a partially structured content. This can be considered one of the thriving forms of software documentation available nowadays [3]. Different to the traditional software documentation, which is produced mostly by a central

*Correspondence: fernanda@doutorado.ufu.br
[1]Faculty of Computing, LASCAM-FACOM, Federal University of Uberlândia, Uberlândia, Brazil
Full list of author information is available at the end of the article

authority, this phenomenon allows anyone to produce and share relevant content for documentation. The content available in such repositories is also known as *crowd knowledge* [4, 5].

An example of the channels that support developer social interaction is Stack Overflow [6], a question-and-answer website where developers collaborate with each other in order to solve issues related to software development. In the Stack Overflow, developers post *questions* related to a programming topic, e.g., an API, while other developers can provide *answers* to help solve their issues [7]. In the context of Stack Overflow, a *question* is an entry consisting of a title, tags and text body, and an *answer* is an entry consisting of a text body (which may include code samples) where a solution, a clarification and/or a in-depth discussion are provided concerning the question.

Stack Overflow has been studied to help researchers to understand the knowledge/mechanisms available on it and how that can be used to assist software development. To investigate the feasibility of using crowd knowledge on Stack Overflow for API documentation, Parnin et al. [8] carried out a study on how content related to a set of APIs is being produced on Stack Overflow. They analyzed the coverage of API elements over *threads*, which is the composition of a question with no or with a collection of answers. However, they did not analyze the nature of threads, which can impact significantly on how suitable threads are for documentation purposes.

The nature of threads can be distinguished by the main concerns of the askers. These concerns are being used in the literature for the definition of *question types* [7, 9, 10]. Examples of question types are *how-to-do-it* – providing a scenario and asking how to implement it—*debug-corrective*—dealing with problems in code already written —*seeking-something*—looking for something objective (e.g., tutorial, tool, library) or subjective (e.g., an opinion, a suggestion, a recommendation)—and *conceptual*—regarding conceptual questions on a particular topic (e.g., definition of concepts, best practices for a given technology). The definition itself of the *how-to-do-it* question type reveals that this type is more adherent to the purpose of documenting how to use API elements. Nonetheless, questions of type *debug-corrective* are still useful as complementary documentation on how to fix frequent problems related to the usage of API elements, while the other types seems to be marginally useful.

In our previous work [11], we conducted a study on the coverage of API elements on Stack Overflow for API documentation, introducing the idea that the coverage analysis should take into account the API documentation purpose. We reported the coverage of Swing API elements by threads with *how-to-do-it* question type in order to measure how much elements are covered in discussions that provide code samples on how to implement a specific task by using the API elements.

In this paper, we extend the previous study, providing coverage analysis of the Swing and Android APIs on threads containing *how-to-do-it* and *debug-corrective* question types. Our main goal is to measure the coverage of API elements to understand how the crowd can contribute for (1) API documentation on *how to use* API elements through code samples to accomplish a specific task given a scenario (coverage on threads containing *how-to-do-it* questions) and (2) API documentation on *how to fix* domain-independent bugs in an existing code where there was a misunderstanding regarding the usage of an API (coverage on threads containing *debug-corrective* questions).

Our overall contribution consists of the intersection of the Parnin et al.'s coverage analysis [8] with two question types defined by Nasehi et al. [9]. Our specific contributions are threefold. First, we developed a classifier by using supervised machine learning algorithms to automatically classify Stack Overflow questions, in order to select threads containing *how-to-do-it* and *debug-corrective* question types. Second, we proposed a methodology for analyzing API coverage on Stack Overflow based on question types, and consequently improving the knowledge for using Stack Overflow content for API documentation. Third, we analyzed the co-occurrence of API elements on threads, i.e., how threads discuss multiple API elements. As complementary analyzes, we investigated (i) the growth of the coverage of API elements as compared to the growth in the number of threads on Stack Overflow related to the same API and (ii) the association between API coverage and its respective usage in a large code base repository.

The remainder of this paper is organized as follows. In the "API coverage by Stack Overflow" section, we positioned our work regarding the work of Parnin et al.. In the "Automatic classification of questions" section, we explain how we classified Stack Overflow questions. And in the "Linking Stack Overflow threads with API elements" section, we explain how we identified API elements on threads. Our methods and experimental setup regarding coverage analysis are presented in the "Methods" section. We present and discuss the obtained results in the "Results and discussion" section, as well as the threats to validity, limitations, and practical implications of the work. In the "Related work" section, we present the related work on (re)documenting APIs, Stack Overflow, and linking documents with code elements. Finally, our conclusions and future work are presented in the "Conclusions" section.

API coverage by Stack Overflow

The analysis of API-related crowd knowledge available on Stack Overflow is essential for providing indicators on how reliable the crowd is at generating as much content as possible for a *complete* API documentation. The *completeness* of the crowd knowledge for an API can be measured by analyzing how many elements of the API are discussed by the crowd, i.e., by *coverage analysis*.

A study on API coverage analysis by the crowd on Stack Overflow was conducted by Parnin et al. [8], which is the work most related to ours. They claimed that a high coverage would suggest that it is feasible to use the crowd knowledge for API documentation as a comprehensive source of knowledge concerning an API.

To analyze API coverage, they built a traceability model between API classes and Stack Overflow threads where these classes are mentioned. Then, they calculated the percentage of API classes linked with at least one thread, resulting in the coverage of API classes. They found that 87.2, 77.3, and 54.3% of the Android, Java, and GWT classes, respectively, are covered by the crowd. They also concluded that despite the potential of a documentation by the crowd to provide many examples and explanations on API elements, the crowd is not reliable for providing content over an entire API.

Their analysis, however, is still too general. They analyzed coverage of API classes with no criterion or filter on Stack Overflow threads regarding the type of API documentation that they target. The type of API documentation can be characterized by the type of content that the documentation should include, which is defined by the intentions from the point of view of the API users, i.e., what they wish to accomplish through its use or knowledge concerning a given API. For example, an intention type can be how to implement specific tasks using an API [5]. Hence, without considering types of API documentation, an understanding of how the API elements are covered by the crowd and how the crowd knowledge can be used for generating API documentation was not possible.

In our work, on the other hand, we have introduced the notion that coverage analysis must be conducted according to the intended type of API documentation. With that in mind, we performed a coverage analysis considering types of Stack Overflow questions related to the main concerns of askers, and thus, subsidizing the choice of threads for different types of API documentation. We took into account the *how-to-do-it* and *debug-corrective* question types:

- *How-to-do-it*: the asker describes a scenario and asks how to implement it (sometimes with a given technology or API) [7, 9]. Figure 1 shows an example of a *how-to-do-it* question.
- *Debug-corrective*: the asker describes or presents problems in the code under development, such as run-time errors, notifications, and unexpected behavior [7, 9]. Figure 2 shows an example of a *debug-corrective* question.

Therefore, the main difference between our work and that of Parnin et al.'s work is that we analyzed the coverage of API elements by the crowd on Stack Overflow considering types of questions for different types of API documentation. We can also point out the following differences:

- We presented an analysis of co-occurrence of API elements on threads, i.e., how threads discuss multiple API elements;
- Instead of analyzing the speed of the crowd at covering API elements over time, as Parnin et al. did, we analyzed the growth of coverage of API elements comparing to the growth in the number of threads on Stack Overflow related to the same API;
- We also analyzed the coverage of API elements with their actual usage, as Parnin et al. did; however, we collected API usage from a large code base repository instead of Google Code Search API (which is no longer available), and we searched for usage in any type of statement instead of searching only in import statements as they did.
- We evaluated our implementation of the linking approach (the identification of API elements in the content of threads), which is based on Parnin et al.'s linking approach and other works [12, 13]. However, as they did not carry out the same evaluation, we cannot judge the reliability of their linking implementation or perform comparisons.
- The Parnin et al.'s coverage analysis relies on API classes, while our coverage analysis relies on three types of API intermediate elements: classes, interfaces, and enumerations.

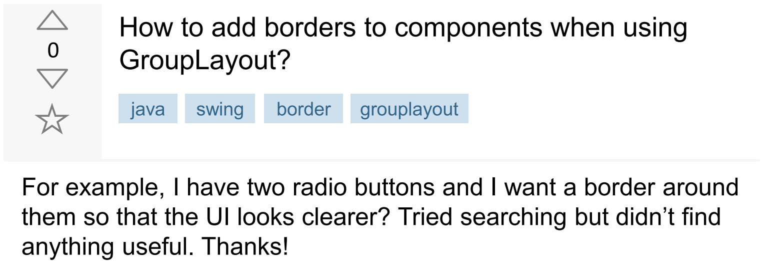
Additionally, we replicated Parnin et al.'s coverage analysis to quantify the difference of coverage considering threads with specific question types (*how-to-do-it* and *debug-corrective*), i.e., our coverage, regarding coverage considering all threads, i.e., their coverage.

Automatic classification of questions

The selection of Stack Overflow threads by specific question types can be characterized as a *text classification problem*. A classification problem consists of mapping a *data sample* (in our case, text) for an appropriate *class* (or label) that is previously known.

Therefore, we rely on supervised machine learning algorithms to classify questions for selecting threads with *how-to-do-it* and *debug-corrective* question types. We could not reuse the classifier built by Souza et al. [7] and Campos et al. [14], as this classifier does not address *debug-corrective* questions. Also the classifier built by Campos and Maia [15] does not focus only on *how-to-do-it* and *debug-corrective*.

The fact that *how-to-do-it* and *debug-corrective* questions are more directly related to API documentation


How to add borders to components when using GroupLayout?

java swing border grouplayout

For example, I have two radio buttons and I want a border around them so that the UI looks clearer? Tried searching but didn't find anything useful. Thanks!

Fig. 1 Example of a *how-to-do-it* question

than *seeking-something* and *conceptual*, the latter are considered as belonging to the *others* class, and so we have a ternary classification problem.

In the remainder of this section, we present the used classification algorithms and tools, the training set construction, the generation of input data for classification, and finally, the evaluation and selection of classification algorithms.

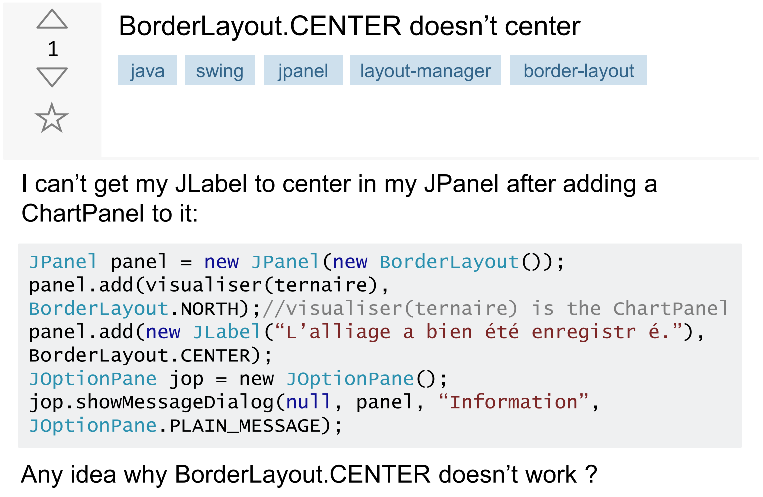
Classification algorithms and tools

There does not exist any classification algorithm that performs better than others for all application domains. For this reason, to select an appropriate algorithm for the classification of Stack Overflow questions, we evaluated and compared the performance of different algorithms, which are listed as follows:

- IBk° (nearest neighbor method) [16]
- J48° (decision tree) [17]
- C45• (decision tree) [18]
- NaiveBayes°• (Bayesian approach) [19]
- BayesNet° (Bayesian approach) [20]
- DecisionTable° (rule-based method) [21]
- MultilayerPerceptron° (neural network) [22]
- SMO° (support vector machine) [23]
- RandomForest° (random forest) [24]
- SimpleLogistic° (linear logistic regression) [25]
- Logistic° (multinomial logistic regression) [26]
- MaxEnt• (maximum entropy) [27]
- MaxEntL1• (multinomial logistic regression with L1 regularization) [27]

These algorithms belong to different types of classifiers. For instance, J48 and C45 are decision tree-based algorithms, while NaiveBayes and BayesNet are Bayesian approaches and SimpleLogistic, Logistic, MaxEnt and MaxEntL1 are logistic regression models. We chose these algorithms because (1) we want to evaluate algorithms from different types of classifiers, (2) they are well-known algorithms, and (3) their implementations are available.

Our evaluation was conducted using Weka [28] and Mallet [29], two open source tools containing a collection of machine learning algorithms, including classification algorithms and support for their evaluation. The difference between these tools is that Mallet is specialized in machine learning applications to text, such as information extraction and document classification, while Weka is for


BorderLayout.CENTER doesn't center

java swing jpanel layout-manager border-layout

I can't get my JLabel to center in my JPanel after adding a ChartPanel to it:

```
JPanel panel = new JPanel(new BorderLayout());
panel.add(visualiser(ternaire),
BorderLayout.NORTH);//visualiser(ternaire) is the ChartPanel
panel.add(new JLabel("L'alliage a bien été enregistr é."),
BorderLayout.CENTER);
JOptionPane jop = new JOptionPane();
jop.showMessageDialog(null, panel, "Information",
JOptionPane.PLAIN_MESSAGE);
```

Any idea why BorderLayout.CENTER doesn't work ?

Fig. 2 Example of a *debug-corrective* question

general machine learning and data mining tasks. In the above list of the algorithms selected for evaluation, ° and • flag algorithms from Weka and Mallet, respectively.

Training set construction

Classification algorithms fall within the category of *supervised* machine learning algorithms. It means that they must be trained with a set of labeled samples (training data) to be able to distinguish unlabeled samples (test data) between a set of classes.

Since we are interested in the selection of Android and Swing threads, we randomly selected 400 questions related to each API from the Stack Overflow database to construct the training sets. We consider that a question is related to an API if some of its tags contain the name of the API ("android" or "swing"). We decided to use non-exact word matching on tags, i.e., to not accept only those which match the exact "android" and "swing" keywords, as we observed that there are Android- and Swing-related questions tagged with packages of the API. For example, the question 951121 has been tagged with "android-widget".

The 800 questions were manually and independently classified by two of the four authors to obtain reliable training sets. At the end of the classification process, we calculate the Kappa statistic [30] for assessing the agreement between the two manual classifications. The observed agreement and the Kappa value for each training set and for both together are presented in Table 1. The observed agreement for Swing (84.75%) was higher than that for Android (78.75%), as the Kappa value as well. For both Android and Swing, individually or together, the strength of the agreements based on Kappa value is considered substantial [30].

Then, the same two authors analyzed only the disagreements to reach a consensus on the correct classification for each question. Table 2 presents the number of mistakes of each author by training set. Interestingly, we observed that some questions were classified differently, but we can accept either of the two classifications, i.e., both classifications are right. In fact, according to Nasehi et al. [9], questions might belong to more than one question type (overlapping).

After the disagreement analysis, we built the final training sets with the correct classifications. We also applied two filters on the training questions. First, we removed the questions where types of question are overlapped. Since the classification maps only one class to a question, we would need to decide which class to assign to questions of overlapped types. Our decision was to discard these, as the training data should be as discriminative as possible. We are not claiming that our training sets are free of questions of overlapped types because they would still exist even if authors 1 and 2 agreed upon a single classification for these. Second, we removed the unanswered questions. Unanswered questions are not useful, and could introduce some sort of misunderstanding into the classifier.

Table 3 presents the numbers of the initial training sets, the numbers of questions of overlapped types and those unanswered (removed), and the numbers of the final training sets. The final training set for Android contains 315 questions, and for Swing 359, totaling 674 questions. Additionally, Table 4 presents the number of questions classified in each of the three classes.

Input data for the tools

The data source used to extract the input data for the tools consists of both the content of questions and the content of one of their answers (the selection of the answer is described in the next paragraph). We used the content of answers since Souza et al. [7] classified Q&A pairs and reported that the answer body provides relevant information to make decision of the Q&A pair class, even considering that the classification applies to questions.

In addition, we used the content of exactly one answer for each question to acquire greater uniformity across the dataset since a question with ten answers has more content than a question with one answer. So, for each question, we selected one of its answers by passing them through a conditional selection sequence: (1) if the question has an accepted answer, select it; otherwise, (2) if there is only one answer with the highest score, select it;

Table 1 Kappa statistic on the training sets built by manual classification

	Observed agreement (%)	Kappa	Strength
Android	78.75	0.675	Substantial
Swing	84.75	0.750	Substantial
Both	81.75	0.712	Substantial

Table 2 Disagreement analysis of the manual classification

	# Author 1's mistakes	# Author 2's mistakes	# Both are right
Android	14	37	34
Swing	10	23	28
Total	24	60	62

Table 3 Filtering of the training sets

	# Questions			
	Initial	Overlap. typ.	Unanswered	Final
Android	400	34	51	315
Swing	400	28	13	359
Total	800	62	64	674

Table 4 Distribution of the training sets' questions in the classes

	# Questions		
Class	Android	Swing	Total
How-to-do-it	112	157	269
Debug-corrective	118	156	274
Others	85	46	131
Total	315	359	674

otherwise, (3) if there is only one answer with the largest code (in bytes), select it; otherwise, (4) the answer with the largest body (in bytes) is selected.

The classification process in Weka uses attribute values extracted from the source data (question plus answer) and in Mallet their textual content is used.

In the case of Weka, we relied on the Souza et al. [7]'s definition of attributes. They defined attributes related to the frequency of predefined terms and expressions (*keywords*) in the content of questions and answers. Each one of these attributes is related to a type of question (*how-to-do-it*, *seeking-something*, *conceptual* and *debug-corrective*), and each keyword has a weight (1 = less important or 5 = more important).

We adopted their keywords of the *how-to-do-it* and *debug-corrective* classes, adding some extra missing keywords. For the *others* class, we used the keywords of the *seeking-something* and *conceptual* classes. Table 5 presents the keywords by class, and those highlighted in bold have weight 5.

We adjusted the keyword weighting method where keywords found in the question title have their weights doubled. This was performed as we observed, during the construction of the training sets, that the asker tends to summarize his main concern in the title of the question.

Moreover, Souza et al. [7] defined one attribute for the frequency of the question mark ("?"), and four boolean attributes for the presence or absence of code or link in the question and answer bodies. We also reused these attributes, and defined four more for the body and code sizes of questions and answers. Table 6 presents those nine attributes used in the construction of the input data for Weka algorithms. The attributes flagged with "*" are the ones reused from Souza et al. [7].

Algorithms evaluation and selection

The classification algorithms were evaluated on three training sets: one built for Android, another one built for Swing, and both together with the aim of choosing which alternative has the best performance.

For Weka algorithms, we defined three different settings for entering the keyword frequencies for the algorithms as follows:

1. The frequencies of the keywords are grouped by class (*how-to-do-it*, *debug-corrective* and *others*). For example, the frequencies of the keywords defined to characterize *how-to-do-it* question type are grouped together. So, we obtain three attributes related to frequencies of the keywords.
2. The frequencies of the keywords are used separately, independent of the class that the keywords were defined as belonging to. The frequencies of the keywords are grouped with the frequencies of their variants instead. For example, the keyword "solve" has the variant "resolve", then their frequencies are grouped into a single attribute.

Table 5 Keywords defined for the three classes

Keywords for *how-to-do-it* class:
how to/do/does/can/i/we, **accomplish**, **achieve**, **implement**, **way(s)**, **anyway**, **(re)solve**, solution(s), step(s), approach(es), function(s), algorithm(s), pseudocode, script(s), manner, mode, workaround, idea(s), suggest, suggestion(s), thought(s), no error(s), no problem(s), is working, work(s/ing) fine/well/ok
Keywords for *debug-corrective* class:
how [...] fix, **not working**, **does not/doesn't/doesnt work**, **did not/didn't/didnt work**, **does not/doesn't/doesnt seem to work**, **nothing works/happens**, **no effect(s)**, **fix**, **bug(s)**, **error(s)**, **exception(s)**, **wrong**, **incorrect**, problem(s), issue(s), mistake(s), missing, trouble(s), debug(ging), fault, fail(ed), unexpected, warning, notice, notification, denied, breakpoint, unhandled, tracker(s), permission(s), weird, strange, crash(es/ed), struggle(ing), does not/doesn't/doesnt
Keywords for *others* class (*seeking* + *conceptual* keywords):
looking/searching for/forward/at/around, **tutorial(s)**, **manual(s)**, **book(s)**, **tool(s)**, package(s), client(s), plugin(s), plug-in, app, application(s), lib(s), library(ies), framework(s), ide(s), article(s), system(s), software, repository(ies), platform(s), video(s), resource(s), technique(s), editor(s), blog(s), debugger(s), interpreter(s), compiler(s), profiler(s), generator(s), guide(s), guidance, guideline(s), orientation(s), direction(s), recommend(ation), suggest, suggestion(s), advice(s), opinion(s), hint(s), point(ers), alternative(s), choice(s), idea(s), thought(s), option(s), clue(s), experience(s), experienced, search(ing), research(ing), google(ing), look(ed), seek(ing), scan(ing), learning, getting, migrate(ing), migration(s), upgrade(ing), convert(ing), conversion, porting, freeware, strategy, started, tips, tricks, caveats, insight(s), light, share, provide, find, material, available, beginner, possibilities, **concept(ual)**, **explain**, **explanation**, **explanatory**, **clarify**, **clarification**, **explicate**, **elucidate**, **illuminate**, **expound**, **practice(s)**, **best practice(s)**, **difference(s) between**, mean(ing), significance, signification, possible, level, metrics, statistics, reason(s), motive(s), cause(s), justification(s), potential, distinction(s), consensus, lesson(s), understand, purpose(s), how much, how many

The keywords in bold have weight 5 and the remaining ones have weight 1

Table 6 Attributes for question types characterization

Attribute name	Description
QUESTION_MARK*	Frequency of the question mark ("?") in the question and answer.
QUESTION_HAS_CODE*	Boolean value indicating whether exists source code in the question.
ANSWER_HAS_CODE*	Boolean value indicating whether exists source code in the answer.
QUESTION_HAS_LINK*	Boolean value indicating whether exists link in the question.
ANSWER_HAS_LINK*	Boolean value indicating whether exists link in the answer.
QUESTION_BODY_SIZE	Bytes value of the question size.
ANSWER_BODY_SIZE	Bytes value of the answer size.
QUESTION_CODE_SIZE	Bytes value of the question code size.
ANSWER_CODE_SIZE	Bytes value of the answer code size.

The attributes flagged with "*" were reused from Souza et al. [7]

Table 7 Settings of the evaluations on Weka algorithms

	Attribute selection	
	No	Yes
Frequencies of keywords grouped by class	1A	1B
Frequencies of keywords grouped with the frequencies of their variants	2A	2B
Frequencies of keywords with information gain grouped by class	3A	3B

3. The frequencies of the keywords are grouped by class as in setting number 1, but taking into account only keywords considered relevant (by the method described below).

Furthermore, Weka algorithms were evaluated twice for each one of those settings: (A) with no attribute selection filter on the training sets and (B) with attribute selection filter. We evaluated the algorithms with a filter of attribute selection for two reasons. First, with a limited amount of training samples, excessive attributes may cause the classifier to overfit the training data, i.e., the learning mechanism fits to peculiarities in the training data and decreases the generalization performance for classifying new samples [31]. Second, irrelevant or redundant attributes may confuse the classification algorithm [32].

We used the information gain method to select the relevant attributes. In this method, each attribute is independently evaluated and a score is assigned to it. Then, the selected attributes are the ones with the score higher than a threshold. In our work, we removed the attributes with score zero. See Appendix "Attribute selection based on the information gain method" for information on the numbers of selected attributes and the attributes with highest information gain value.

Table 7 summarizes the settings of the evaluations on Weka algorithms. For future reference, "1A", for example, refers to a test where the frequencies of the keywords were grouped by class and no attribute selection filter was applied to the attributes.

The evaluation of each algorithm was performed using 10-fold cross-validation, where the training data is split in k equal parts (folds), and for each kth iteration, the classification algorithm is trained with $k - 1$-folds and tested with the remaining one [33]. At the end of the process, fold accuracies (correct classification rates) are aggregated by average calculation.

Table 8 presents, by tool (Weka and Mallet), the classification algorithms and their respective evaluation results (overall accuracy) on the three training sets. The highest accuracy obtained for each training set is highlighted in bold. For presentation reasons, we do not present all accuracies obtained by Weka algorithms with the six different settings. Only the highest accuracy of each algorithm on each training set is presented, flagged by the setting in which it was obtained (see Table 7 for the meaning of the flags).

For the Android training set, the best algorithm was SimpleLogistic with an overall accuracy of 78.45%, and for the Swing training set it was Logistic with 78.83%. The best accuracy for both Android and Swing together was 78.19%, obtained with the Logistic algorithm. Due to the

Table 8 Evaluation of the classification algorithms: overall accuracies based on 10-fold cross-validation on the training sets of the Android and Swing

Tool	Classification algorithm	Android (%)	Swing (%)	Android + Swing (%)
Weka	IBk	73.35^{3B}	76.33^{3B}	74.79^{1A}
	J48	73.02^{1B}	78.56^{1A}	73.31^{3B}
	NaiveBayes	68.21^{2B}	74.66^{2A}	71.67^{3B}
	BayesNet	73.35^{23AB}	75.23^{1AB}	74.65^{1AB}
	DecisionTable	70.83^{3AB}	77.73^{3AB}	73.02^{1B}
	MultilayerPercep.	73.97^{3B}	76.06^{1B}	76.44^{1B}
	SMO	72.7^{3B}	75.24^{2A}	75.98^{2B}
	RandomForest	73.66^{1A}	78.02^{1A}	75.54^{3A}
	SimpleLogistic	*78.45*^{2B}	78.28^{2B}	77.46^{2B}
	Logistic	77.5^{3B}	*78.83*^{2B}	*78.19*^{2B}
Mallet	C45	57.14	53.47	54
	NaiveBayes	67.24	64.94	63.24
	MaxEnt	71.76	69.37	76.58
	MaxEntL1	77.5	69.95	77.01

fact that there is no advantage gained using both training sets together, we chose to select the best algorithm for each API in order to classify its questions.

We also observed that some algorithms have similar performance, in terms of accuracy, with the SimpleLogistic algorithm for Android and with the Logistic algorithm for Swing. For example, for Android, the SimpleLogistic algorithm had an overall accuracy of 78.45% and the Logistic and MaxEntL1 algorithms had 77.5% of accuracy. To check if the performance of these algorithms are significantly different, we used McNemar's test [34, 35]. This test employs the z score equation, which uses the number of samples where one of the algorithms failed and the other succeeded, i.e., the performance discrepancies.

We calculated the z score for each API considering the algorithm with the highest accuracy and the algorithms with accuracy 1% lower than the highest ranking algorithm. When the z score value is equal to 0, the two algorithms are said to show similar performance, and when the z score diverges from 0 in positive direction, this indicates that their performance differs significantly [35].

We concluded that, for Android, the performance of the SimpleLogistic algorithm differs significantly from the performance of the Logistic ($z\ score = 0.38$) and MaxEntL1 ($z\ score = 0.23$) algorithms, thus we chose to use SimpleLogistic to classify Android questions. For Swing, the performance of the Logistic algorithm differs significantly from the performance of the SimpleLogistic ($z\ score = 0.2$) and RandomForest ($z\ score = 0.26$) algorithms, but it is similar to the performance of the J48 algorithm ($z\ score = 0$). It means that we could choose any of the two algorithms (Logistic or J48) to classify Swing questions, and so we opted for the Logistic algorithm.

In addition, Tables 9 and 10 present the performance of the selected algorithms (SimpleLogistic and Logistic, respectively) for the APIs (Android and Swing, respectively) by class. Noted here through an analysis of the f-measure is that for both APIs, the selected algorithm performs better at classifying *debug-corrective* questions but shows an inferior performance when classifying *others* questions.

Linking Stack Overflow threads with API elements

The coverage analysis of API elements by the crowd on Stack Overflow requires the creation of links between Stack Overflow threads and API elements. Our linking approach style relies on three works [8, 12, 13]: from a list of elements of a given API, the names of these elements are searched for within content of given threads (related to the API), and when a match is found, a link is created between the thread and the API element. At the end of the process, each API element has a list of threads where its name is mentioned.

Table 9 Performance of the SimpleLogistic algorithm for Android by class

Class	Precision	Recall	F-measure
How-to-do-it	0.824	0.821	0.822
Debug-corrective	0.804	0.864	0.833
Others	0.733	0.625	0.675

The search for API elements is performed by their *short names*, i.e., their names without their respective package names. The short name of a top-level element is its non-qualified name, i.e., the single name of the element. For instance, the short name of the element `javax.swing.JFrame` is `JFrame`. The short name of an inner element, on the other hand, is its non-qualified name preceded by the non-qualified name of its top-level element plus a dot. For instance, the short name of the element `javax.swing.JFrame.AccessibleJFrame` is `JFrame.AccessibleJFrame`.

In the remainder of this section, we present the link types considered for linking threads with API elements, the preprocessing of threads, the identification of API elements in Stack Overflow posts strategy, and finally, the evaluation of the linking approach.

Link types

There are different types of textual content on Stack Overflow posts in which the name of API elements can be mentioned [8, 12, 13], called *link types*. We adopted the four link types from Parnin et al. [8], with some minor modifications on how the name of API elements may match in the content of some link types. The link types are described as follows:

Code sample link: A match for an API element name occurring in the text inside `<code></code>` tags;

Code markup link: A match for an API element name partially enclosed by `<code></code>` tags;

Href markup link: A match for an API element name occurring in the text inside `<a></a>` tags;

Word link: A match for an API element name occurring in the text outside of `<code></code>` and `<a></a>` tags.

Table 10 Performance of the Logistic algorithm for Swing by class

Class	Precision	Recall	F-measure
How-to-do-it	0.788	0.795	0.791
Debug-corrective	0.832	0.885	0.858
Others	0.695	0.425	0.527

Threads preprocessing

Before performing the linkage of threads with API elements, we preprocessed the content of threads. This preprocessing consists of separating the content of each post of threads (question and answer) according to the link types. We used Jericho HTML Parser [36] to perform this task.

Given a post, for code sample and code markup links, we first collect all `<code>` elements existing in the post. Then, we verify if there exists white-space between strings in the content of each `<code>` element. In those cases where it does not occur, we consider the content of the `<code>` element as code markup. Otherwise, that content is separated for code sample link.

For href markup link, we just collect all `<a>` elements existing in the post. For word link, we removed the content of `<code>` and `<a>` elements from a copy of the whole post body.

We also removed comments and natural language texts contained in the code samples (content separated for code sample link). We used the regular expressions numbers 1 and 2 presented in Table 11 for finding comments and texts.

Table 11 Regular expressions used in the process of linking Stack Overflow threads with API elements

1) `((?s)(/.*?/))|(//.*)`
   Regex for matching comments inside "/* */" or preceded by "//"

2) `((?s)(".*?"))`
   Regex for matching natural language text inside ""

3) `(\bElementShortName\b)`
   Regex for API element short name boundary matching. For example: the `regex (\bJCheckBox\b)` matches `JCheckBox` but does not match `JCheckBoxMenuItem`

4) `(\bElementShortName\b)(?!.PartialShortNames)`
   Regex for API element short name boundary matching with negative look-ahead. For example: the regex `(\bDefaultTableCellRenderer\b)(?!.UIResource)` matches `DefaultTableCellRenderer` but does not match `DefaultTableCellRenderer.UIResource`

5) `(?<!PartialShortNames.)(\bElementShortName\b)`
   Regex for API element short name boundary matching with negative look-behind. For example: the regex `(?<!ScrollPaneLayout.|BasicComboBoxEditor.)(\bUIResource\b)` matches `UIResource` but does not match neither `ScrollPaneLayout.UIResource` nor `BasicComboBoxEditor.UIResource`

6) `(\b([A-Z_]+[a-z_0-9]+)2,\b)|(\b([A-Z_])2,[a-z_0-9]+\b)|([A-Z_]+[a-z_0-9]+[A-Z_]+\b)`
   Regex for matching multi-word API element short name (camel case). For example: the regex matches `SwingUtilities`, `JFrame`, `HTMLDocument` and `TextUI`, but does not match `View` and `HTML`

Identification of API elements in Stack Overflow posts

For creating links between Stack Overflow threads and API elements, it is necessary to identify mentions of the API elements in the (preprocessed) content of thread posts. So, for each thread, the mention of API element short names (names without package names) are searched for within the content of its posts by regular expressions (regex) based on *word boundary matching*.

A word boundary can be defined as a position where a word character (normally letter or digit) is followed or preceded by a non-word character (such as white-space). For instance, the regex `(search)` matches the "search" word, but also matches the "research" and "searcher" words. For finding only exact matches for the "search" word, word boundaries must be used in the regex, which are specified by `\b((\bsearch\b))`.

We used word boundaries for finding matches of API element short names in Stack Overflow posts to ensure that these names occur at the beginning and the end of a sequence of characters. We used the regex number 3 presented in Table 11 for performing this task.

Moreover, there are three situations where threads can be mistakenly linked with API elements. These situations, and how we handled these to avoid false positives, are presented as follows.

First, since we search for API elements by their short names, we could acquire false positives by linking threads with API elements that share the same short name. For such elements, we check if their package names appear in the threads where their short names matched. If the package name of an API element is found, the thread is linked with that API element. Otherwise, i.e., if no package of those API elements is found, the thread is not linked with any of those API elements.

Second, the existence of inner elements in the API raises a problem of matching API elements using word boundaries, as the dot character is considered a non-word character. This problem occurs in two forms. Form A: the name of outer elements also matches the name of their inner elements. For example, at searching the `DefaultTableCellRenderer` element name by using the regex number 3, the `DefaultTableCellRenderer.UIResource` name will also be a match. Form B: the name of (inner or non-inner) elements can also match with the name of inner elements. For example, when searching the `UIResource` element name, the `ScrollPaneLayout.UIResource` and `BasicComboBoxEditor.UIResource` element names will also be matches.

To avoid this problem, we used negative look-ahead for form A and negative look-behind for form B. Negative look-ahead and look-behind aim at ensuring that the pattern of interest is not immediately followed or preceded by another pattern, respectively. In our case, regarding the `DefaultTableCellRenderer` element name, we used negative look-ahead (regex number 4 in Table 11) for ensuring that the name is not followed by `.UIResource`. In a similar way, we used negative look-behind (regex number 5 in Table 11) in order to ensure that the `UIResource` name is not preceded by `ScrollPaneLayout.` and `BasicComboBoxEditor.`.

Third, one-word API element short names, like `View`, can be confused with English words in natural language texts. To avoid false positives regarding this issue, we only search for multi-word API element short names in the content of word and href markup links. The regex 6 was used to check if an API element short name consists of a multi-word name.

Linking approach evaluation

We evaluated the implementation of our linking algorithm for Stack Overflow threads with API elements to obtain an indicator on how reliable it is. We randomly selected 50 threads, 25 related to each API.

First, we manually analyzed each thread and identified the API elements that are mentioned on it. We found 129 Swing and 134 Android element occurrences in these 50 threads by the manual analysis, totaling 263 API elements occurrences. Second, we executed the linking algorithm on these threads.

Then, for each thread, we verified if the API elements linked with respective threads are the same API elements identified manually. The overall precision was 99.20% and recall was 94.68%. We observed that our linking algorithm is not able to identify occurrences of API elements when an API element is not mentioned exactly by its short name. For instance, in the thread containing the question 9785173, the Android element `android.widget.BaseAdapter` was mentioned as “Base Adapter”. This type of occurrence of API element is missed by the linking approach, and consequently the thread is not linked with the API element.

Analysis of actual semantic links

One characteristic of our linking approach used for coverage analysis is that we do not deepen on the *type of occurrence* of an API element in a Stack Overflow post. Our coverage results (which are presented in the “Results and discussion” section) are based on the analysis of any type of occurrence of API elements in the content of threads. In other words, those threads may be *potentially* useful to document those API elements based on their nature: *how-to-do-it* or *debug-corrective*. However, the occurrence of an API element in the content of a thread does not *necessarily* mean that such discussion, or part of it, is in fact useful for documenting that element. There would be different types of API element occurrences that would require a semantic analysis of the post content to define the respective type of API element occurrences.

Due to the fact that our linking approach does not work at a semantic level, we cannot assure with certainty the exact number of API elements that are actually semantically covered for either *how-to-do-it* or *debug-corrective* posts. Nonetheless, in this section we perform a manual and qualitative study in order to understand a possible decrease in our coverage results when the semantics of Stack Overflow posts is taken into account.

The challenge of discovering if the content (or part of it) of an informal documentation, as Stack Overflow posts, is pertinent to API elements was addressed in the works of Rigby and Robillard [37] and Petrosyan et al. [38]. Rigby and Robillard proposed a classification-based solution to detect which code elements in a document are salient to the topic of the Stack Overflow post—“a code element is *salient*, if it is central to an example code fragment or if there is some discussion defining its function or describing its use” [37]. Petrosyan et al. [38], on the other hand, proposed a classification-based solution to discover tutorial sections that explain a given API type (classes and interfaces), i.e., *relevant* tutorial sections to API types. They considered that a tutorial section is relevant to an API type if it would help a reader unfamiliar with the corresponding API to decide when or how to use the API type to complete a programming task.

In the context of our work, however, the concepts of *salient code element* [37] and *relevant tutorial sections to API types* [38] are not exactly adequate. By these definitions, even if an API element mentioned in a Stack Overflow post is not salient, or even if a tutorial section (a thread, in our case) is not relevant to an API element, it does not mean that there is no useful content there for documenting the element.

In this sense, we define what is *relevant for documenting*: a thread is *relevant for documenting* an API element if it contains any relevant content for a *specific type of documentation* of that API element. Since we are interested in two types of documentation, i.e., documentation containing *how-to-do-it* and *debug-corrective* content, and the nature of threads containing these two types of discussions are different, we defined that a thread is relevant for documenting an API element as follows:

- a thread of *how-to-do-it* nature, or a part of it, is considered relevant for documenting an API element if there is any example code fragment in an answer demonstrating the use of the element, even when it is not the central point of the thread;

- a thread of *debug-corrective* nature, or a part of it, is considered relevant for documenting an API element if the element is part of the central point of the bug reported by the asker (where the asker thought that the bug had occurred) or the central point of the bug fixing (where the bug actually occurred).

Then, to quantify a potential decrease in our coverage results when the semantics of Stack Overflow posts is taken into account, we manually analyzed a sample of threads to investigate in how many cases there exists content for documenting API elements in *how-to-do-it* and *debug-corrective* discussions. We randomly selected 200 threads for each API, where 100 were classified as *how-to-do-it* and the other 100 as *debug-corrective*, totaling 400 threads.

Each of those 100 threads were not sampled completely randomly—we noted the need of selecting threads that mention API elements of different ways, so we decided to perform a stratified random sampling based on previous data analysis. First, we decided to select 30% out of those 100 threads to compose group 1 of samples:

Group 1: threads that are unique for API elements, i.e., threads that mention API elements covered only by one thread.

It is important that threads contained in group 1 to be part of our sample as we need to be sure that we also assess if that only one thread citing a given API element is relevant for its documentation. Second, we decided sampling the remaining 70% of threads by the amount of API elements that they cover, composing groups 2 and 3:

Group 2: threads that cover only one element of the API under analysis.

Group 3: threads that cover multiple elements of the API under analysis.

Threads that cover only one element of the API under analysis do not necessarily concern that API. For instance, a thread that covers only one Android element can be, actually, about another API. Therefore, we selected these threads to investigate how many of these were tagged as being on Android and Swing but are actually not on those APIs. As we noted, from Table 23, there are a small percentage of threads that cover only one element of the API under analysis. Based on the Table, for *how-to-do-it* and *debug-corrective* threads, we decided to select 70 and 85% of them, respectively, as threads that cover multiple API elements. Table 12 presents a summary of the numbers of threads and Table 13 presents a summary of the numbers of occurrences of API elements (in those threads) contained in the groups of samples.

The samples were manually and independently analyzed by two of the four authors to obtain more reliable results. At the end of the analysis process, we calculate the Kappa statistic [30] for assessing the agreement between the two manual analyses. The observed agreement and the Kappa value for each type of discussion are presented in Table 14. The strength of the agreements based on the Kappa value is considered substantial for *debug-corrective* and slight for *how-to-do-it* [30]. The observed agreement, however, for both *debug-corrective* (91.32%) and *how-to-do-it* (88.73%), were high.

The same two authors analyzed only the disagreements to reach a consensus on the correct answer for each occurrence of API element in the threads, i.e., if a given thread contains any relevant content for a type of documentation of a given API element or not. In this analysis, we observed that the task of finding relevant content for documenting specific API elements is not trivial: sometimes it can be subjective. For example, we found a reason for why the strength of the agreements concerning *how-to-do-it* was slight: one of the authors did not consider XML code as an example code fragment in an answer that demonstrates the use of an API element. As an example, the thread containing the question #15751365, entitled "android how to code dialog layout", contains an answer that demonstrates the use of the `android.widget.RelativeLayout` element in an XML code, but one author marked that thread as not being relevant for documenting that element. In the discussion of the disagreements, the two authors decided to consider XML code as an example of code fragment. Concerning *debug-corrective* related threads, the two authors also have some difficulties to reach a consensus. One of those cases was to understand which are the actual API elements that the asker thinks that the bug/problem is. For instance, in the analysis of the thread containing the question #7549887, entitled "Can't figure out how to overlap images in java", one author marked that `javax.swing.JPanel` element was involved in the problem reported by the asker, and the other author marked `javax.swing.JFrame` element. In the disagreement analysis, the authors agreed that both elements were involved.

After the agreement analysis, we obtained the final results of the manual analysis with the correct answers.

Table 12 Summary of the numbers of threads contained in the groups of samples, for each APIs, for both types of discussion

	# Threads			
Discussion type	Group 1	Group 2	Group 3	Total
How-to-do-it	30	21	49	100
Debug-corrective	30	10	60	100

Table 13 Summary of the numbers of occurrences of API elements contained in the groups of samples, for both APIs and both types of discussion

		# Occurrences analyzed			
API	Discussion type	Group 1	Group 2	Group 3	Total
Swing	*how-to-do-it*	36	21	244	301
	debug-corrective	30	10	353	393
Android	*how-to-do-it*	33	21	204	258
	debug-corrective	32	10	452	494

Table 15 presents the results regarding the proportion of occurrence of API element that the analyzed threads are relevant for documenting such. For both APIs, there exists relevant *how-to-do-it* content for documenting a high number of occurrence of API elements—the total percentages of occurrences that threads are actually relevant for are 94.35% for Swing and 86.82% for Android. *Debug-corrective* content, on the other hand, is not relevant for documenting a high number of occurrence of API elements—there exists relevant *debug-corrective* content for only 40.71 and 23.89% of the occurrences of Swing and Android elements, respectively.

However, concerning coverage of API elements, those proportions are not exactly adequate to quantify how much coverage can potentially be decreased from our coverage based on our non-semantic linking approach. Those proportions are based on the occurrences of API elements, which means that a given API element can occur in more than one thread. So, even if a given thread were not considered relevant for documenting that element, it does not mean that there is no other thread relevant for such element.

To perform that quantification, for each API element mentioned in the threads of our sample, we searched for at least one occurrence in a thread that was considered relevant for documenting it. Then, we calculated the percentage of API elements that we found relevant content for documenting them. Table 16 presents the numbers of API elements contained in the sample and the overall percentage of API elements actually contained in relevant content.

Based on the sample estimate results, we can note that the percentages of API elements mentioned in *how-do-to-it* related threads and that are actually contained in relevant content for their documentation are high—there exists content in the sampled threads for almost 90% of the Swing and Android elements. Regarding *debug-corrective* content, on the other hand, the decrease from the amount of API elements mentioned in the threads to the amount of API elements contained in relevant content of such threads is higher—there exists content in the sampled threads for only 50% of the Swing elements and for only 43.3% of the Android elements. Although these numbers show that there is a higher decrease on the coverage for *debug-corrective* documentation than for *how-to-do-it*, the numbers should not be used absolutely to infer the increase in the nominal coverage results we will show for RQ #1 and RQ #2, as the decrease tends to be lower as we enlarge the sample.

Table 14 Kappa statistic on the manual analysis

	Observed agreement (%)	Kappa	Strength
How-to-do-it	88.73	0.119	Slight
Debug-corrective	91.32	0.796	Substantial

Table 15 Proportion of occurrence of API element which threads are relevant for documenting such, for both APIs and both types of discussion

API	Discussion type	Relevant for documenting (%)
Swing	*how-to-do-it*	94.35
	debug-corrective	40.71
Android	*how-to-do-it*	86.82
	debug-corrective	23.89

Methods

Our main goal is to analyze Stack Overflow discussions for the purpose of understanding how the crowd can contribute to document APIs, with respect to coverage of API elements in threads for *how-to-do-it* and *debug-corrective* tasks. To achieve our goal, we formulated five research questions:

RQ #1. To what extent does Stack Overflow cover API elements of Swing and Android APIs with threads containing *how-to-do-it* question? This research question aims at providing an indicator on how well API elements are covered by threads that can be potentially useful to document these elements regarding *how-to-do-it* tasks.

RQ #2. To what extent does Stack Overflow cover API elements of Swing and Android APIs with threads containing *debug-corrective* question? This research question

Table 16 Results on API elements actually contained in relevant content of the sampled threads, for both APIs and both types of discussion

API	Discussion type	# Elements	Relevant for documenting (%)
Swing	*how-to-do-it*	115	89.57
	debug-corrective	118	50
Android	*how-to-do-it*	144	89.58
	debug-corrective	194	43.3

aims at the same as RQ #1, but regarding *debug-corrective* tasks.

RQ #3. How often do threads cover multiple API elements? This research question aims at providing an indicator on how many threads are dedicated for only one specific API element, and how many threads cover multiple API elements. Moreover, it aims at providing indicators on API elements that are frequently mentioned together. This analysis can provide insights on API elements that must be documented together.

RQ #4. How does Stack Overflow cover API elements over time? This research question aims at investigating the growth of coverage of API elements compared to the growth of threads related to the API over time. This analysis can provide an indicator of the maximum coverage that the crowd can reach for an API whether Stack Overflow reaches saturation when covering API elements.

RQ #5. Is there an association between coverage of API elements on *how-to-do-it* and *debug-corrective* discussions by the crowd and actual usage of these elements in software systems? This research question aims at verifying whether API elements covered by a high number of threads are used in a high number of projects, and API elements covered by a low number of threads, or not covered, are used in a low number of projects, or not used at all. This analysis can provide an explanation for API elements that are not discussed by the crowd.

Data collection

The necessary data for answering the research questions are (1) the Stack Overflow threads on Swing and Android, (2) a list of classes, interfaces and enumerations existing in Swing and Android, and (3) usage references of Swing and Android elements in source code of real software projects.

1) We downloaded the Stack Overflow threads from the Stack Exchange Data Dump [39], release of January 2014 of the Stack Overflow public data dump, and we imported these into a relational database to facilitate queries. Specific threads for Swing and Android were selected in the same way as shown in one of the previous sections, i.e., by searching for threads where their questions have a tag containing the name of the API ("android" or "swing").

2) We obtained the lists of Swing and Android intermediate elements from *javadoc jar* files. For Swing, we obtained the jar file from the Java official documentation [40], containing 923 Swing intermediate elements: 823 classes, 90 interfaces and 10 enumerations, distributed in 18 packages. For Android, we obtained the jar file from The Central Repository [41], containing 1678 Android intermediate elements: 1259 classes, 361 interfaces and 58 enumerations, distributed in 66 packages. It is worth mentioning that these numbers of API elements include top-level and inner elements. A top-level element is a direct member of a package, while an inner element is defined as a member inside another element. Since inner elements are public, they are also part of the API of a library or framework, so we consider them in the coverage analysis. Also, we selected only elements in packages that begin with "javax.swing" and "android", for Swing and Android, respectively, although those *jar* files contained other utilities.

3) We obtained the usage of Swing and Android elements from the Boa infrastructure [42], which provides the metadata for almost 8,000,000 GitHub projects. Projects identified as Java projects also include repository history and source code. The source code of a Java file is stored as abstract syntax tree (AST). Boa provides a syntax inspired by the object-oriented visitor pattern to analyze projects and their ASTs using depth-first search traversal strategy. In a visit function, visit clauses are defined to visit AST node types, and when the visitor visits any node matching the specified node type, the body of the clause is executed.

We coded a visit function for collecting Swing and Android usage, which traverses the ASTs of full development histories of Java projects. The types of AST node of interest are Project, ASTRoot and Type ones. We collected the name of the projects in a visit clause defined for Project node. API usage information is obtained in the visit clauses defined for the ASTRoot and Type nodes. In ASTRoot nodes, we searched imports (generic or not) of the APIs. In Type nodes, we searched (i) fully-qualified name of types (classes, interfaces and enumerations) used in statements and (ii) any other types used in statements of an AST where an import of the API was found. For finding imports and fully qualified name of types, we check if their names start with "javax.swing" and "android", for Swing and Android, respectively. We post processed the non-qualified types found in ASTs where an import of the API was also found. We check whether these types belong to any package found as a generic import in the same AST. Types not found in any these packages are discarded.

Thread selection based on question types

To select threads containing *how-to-do-it* and *debug-corrective* question types for coverage analysis, we performed the classification of the collected questions by using the classifiers developed in the "Automatic classification of questions" section. We trained the SimpleLogistic and Logistic algorithms with the Android and Swing training sets, respectively. Then, we filtered questions used in the training sets from the initially collected set of questions. This filtered set was preprocessed to generate the input data files for the classifiers and finally, we ran the classifiers with the respective API input data files

(Android files for SimpleLogistic and Swing files for Logistic). Table 17 presents the number of questions for each API by class according to the produced classification.

Noted here is that the difference of proportions between Swing and Android in the distribution of the classified questions in the classes is mainly caused by the fact that a larger fraction of questions from Android API was classified as *others*. Moreover, we note that there are more questions classified as *debug-corrective* than *how-to-do-it* for Android. Differently, the proportions of questions classified as *debug-corrective* and *how-to-do-it* are pretty close for Swing.

It is worth mentioning that the thread selection based on question types for coverage analysis consider those questions classified and presented in Table 17, along with those in the training sets. For example, for Android coverage analysis that only considers threads with *how-to-do-it* questions, the input threads for coverage analysis consist of 151,843 (see Table 17) plus 112 (see Table 4), totaling 151,955 threads. Table 18 presents the total number of questions by class, which were selected for coverage analysis.

API coverage analysis

For coverage analysis, we propose some filters on the Stack Overflow threads. First, threads are filtered on their respective type of analysis (*how-to-do-it* or *debug-corrective*). Moreover, for *how-to-do-it* questions, we defined that for an API element to be considered as covered, it should necessarily be mentioned at least once in a *code sample* of an *answer* of a thread. We defined these two filters based on the nature of the *how-to-do-it* question type, in which the questioner provides a scenario and asks about how to implement it. For the specific documentation purpose of explaining how to use API elements, we argue that API elements should necessarily be mentioned in *code samples* of *answers* to be considered covered as follows. If an API element is mentioned in the question but is not mentioned in an example (code samples) in the answer, the element is not part of the solution of the scenario provided. Then, that thread seems to have little relevance for *how-to-do-it* documentation [5], so the API element should not be considered covered by the thread.

Table 17 Distribution of the classified questions in the classes

	# Questions	
Class	Android	Swing
How-to-do-it	151,843 (35.79%)	17,426 (45.87%)
Debug-corrective	180,577 (42.56%)	17,391 (45.78%)
Others	91,879 (21.65%)	3169 (8.34%)
Total	424,299	37,986

Table 18 Distribution of all questions in the classes

	# Questions	
Class	Android	Swing
How-to-do-it	151,955	17,583
Debug-corrective	180,695	17,547
Others	91,964	3215
Total	424,614	38,345

Figure 3 provides an overview of the coverage analysis process. The input is a list of Stack Overflow threads and a list of API elements. After the filters are applied on threads, we linked the threads with the API elements by using the approach detailed in the "Linking Stack Overflow threads with API elements" section.

Observe that there are three entries for coverage analysis regarding Stack Overflow threads. The dotted flow indicates the preprocessing of the threads for coverage analysis on *how-to-do-it* questions, i.e., the *thread filter* to select threads with *how-to-do-it* question, the *post filter* to discard the questions from the threads, and the *content filter* to select the code samples of the answers. The dashed flow indicates the preprocessing of the threads for coverage analysis on *debug-corrective* questions, i.e., the *thread filter* to select threads with *debug-corrective* questions. Finally, the dashed and dotted flow illustrates Parnin et al.'s coverage analysis approach [8], where no preprocessing on threads is performed.

Results and discussion

In this section, we present the results for the posed research questions.

How-to-do-it coverage of API elements

To answer RQ #1, we performed coverage analysis on the threads containing questions classified as *how-to-do-it*: 17,583 threads related to Swing and 151,995 threads related to Android. Only code samples in answers of these threads were analyzed. Table 19 presents the coverage results. Noteworthy here is that almost half of the threads of both APIs was linked with some API element: 47.77% of the Swing threads and 46.28% of the Android threads.

For Swing, we observed that there is at least one answer containing code sample to a *how-to-do-it* question for 40.83% of the classes, 83.33% of the interfaces, and 70% of the enumerations, totaling 45.29% of the Swing elements. The coverage values of interfaces and enumerations are reasonably high, but not even half of the Swing classes are covered by the crowd regarding the purpose of documenting on how to use API elements by code samples to accomplish a specific task. This may suggest that a large part of the API may either be very easy to use dismissing

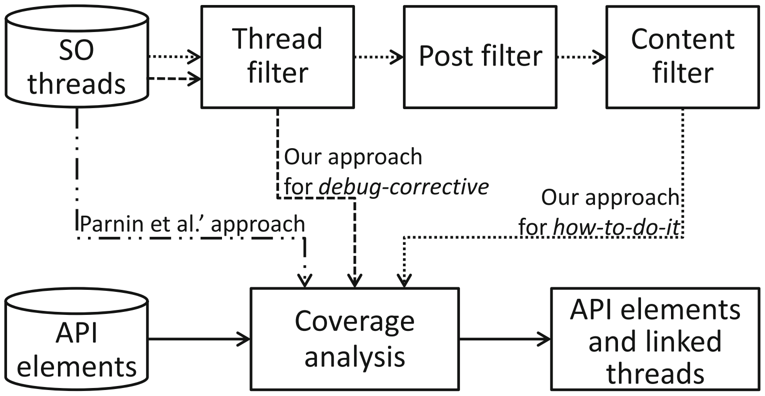


Fig. 3 Overview of the coverage analysis process

further support or that it is barely used by developers in general.

Moreover, we observed that among all the Swing classes, 43.38% are inner classes. Only 14.29% of these are covered, and 62.83% of the uncovered classes are inner ones. Considering only top (non-inner) classes, the coverage is 20% higher (61.16%). Although there are less interfaces and enumerations than classes, the coverage for inner elements of those elements is also low. Only 25% of the inner interfaces are covered, and 60% of the uncovered interfaces are inner. For enumerations, the three uncovered ones are inner enumerations.

For Android, we observed that there is at least one answer containing code sample to a *how-to-do-it* question for 71.17% of the classes, 64.27% of the interfaces, and 58.62% of the enumerations, totaling 69.25% of the Android elements. These coverage values indicate that classes are better covered than interfaces and enumerations in Android.

Regarding the inner elements, among all the Android classes, 27.96% are inner classes and 58.52% are covered. Moreover, 40.22% of the uncovered classes are inner ones. Considering only top classes for Android, we observed that the coverage is higher (76.07%), but it is not much higher than the overall class coverage. In regards to interfaces, among all the Android interfaces, 74.52% are inner and 58.74% are covered. Moreover, 86.05% of the uncovered interfaces are inner. For enumerations, all the uncovered ones are inner enumerations.

We also analyzed the coverage of elements intra-package. The results for Swing are presented with different perspectives in Fig. 4 and Table 20, column "Coverage/How-to-do-it". We observed that two packages (out of 18) are completely covered: `javax.swing.filechooser` and `javax.swing.text.rtf`, which are also the smallest. Considering the two largest packages, `javax.swing` has an average coverage (63.07%), and `javax.swing.plaf.basic` is poorly covered (18.13%). Moreover, there is one package completely uncovered (`javax.swing.plaf.multi`), which is represented by the scratched rectangle.

To verify whether the size of package influences coverage, we calculated Spearman's rank correlation coefficient (ρ) between package size and coverage. Correlation coefficients measure the extent to which two variables tend to change together. In the case of ρ, a positive value indicates that coverage tends to increase when package size increases, and a negative value indicates that coverage tends to decrease when package size increases. For the sizes and the *how-to-do-it* coverage of the Swing packages, we found $\rho = -0.33$, which means a weak negative correlation, i.e., the coverage tends to decrease when package size increases.

Table 19 Coverage results on Swing- and Android-related threads containing *how-to-do-it* question type

API	# Threads	# Threads	Coverage considering all elements				Coverage considering only non-inner elements			
	analyzed	linked	Classes	Interfaces	Enums	Total	Classes	Interfaces	Enums	Total
Swing	17,583	8400	40.83%	83.33%	70%	45.29%	61.16%	92.31%	100%	65.75%
			336	75	7	418	285	72	2	359
Android	151,955	70,325	71.17%	64.27%	58.62%	69.25%	76.07%	80.43%	100%	76.52%
			896	232	34	1,162	690	74	2	766

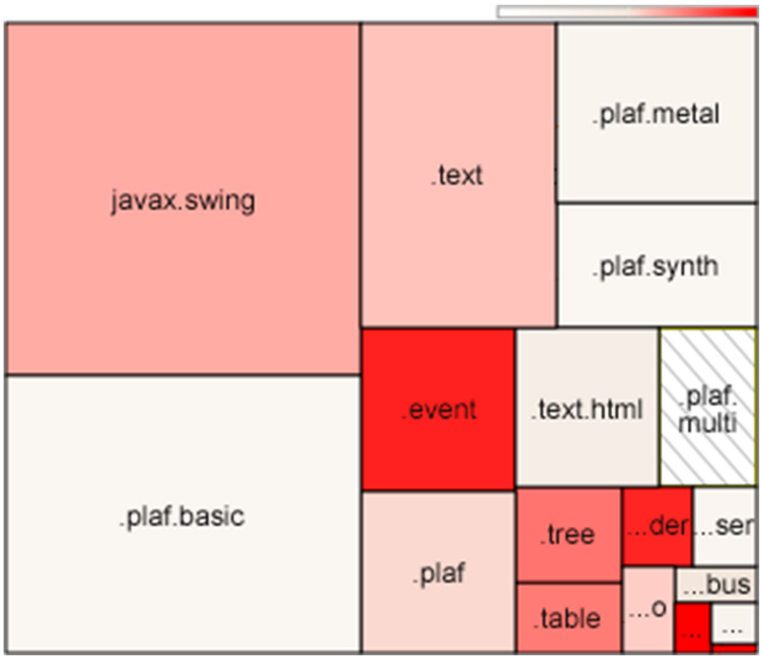


Fig. 4 Swing intra-package coverage for *how-to-do-it* tasks considering the three API element types. The color scale represents percentage values of coverage of the elements—*white* represents the lowest value > 0% and *red* represents 100%. *Scratched rectangle* represents 0%. The *size* of each rectangle is proportional to the number of existing elements in the package

Table 20 Swing intra-package coverage considering the three API element types

		# Threads linked		Coverage (%)	
Package	# Elements	*How-to-do-it*	*Debug-corrective*	*How-to-do-it*	*Debug-corrective*
javax.swing	241	7932	16,458	63.07	68.05
.plaf.basic	193	135	445	18.13	27.98
.text	116	523	952	57.76	68.1
.plaf.metal	70	35	66	22.86	28.57
.plaf.synth	50	12	48	18	50
.event	49	571	1205	91.84	95.92
.plaf	49	114	206	53.06	69.39
.text.html	44	66	143	36.36	50
.plaf.multi	31	0	1	0	3.23
.tree	20	159	243	75	70
.table	15	757	1748	73.33	86.67
.border	11	382	998	90.91	100
.text.html.parser	10	7	53	20	60
.undo	9	3	9	55.56	66.67
.plaf.nimbus	6	25	68	50	83.33
.colorchooser	4	6	5	25	75
.filechooser	4	58	138	100	100
.text.rtf	1	1	8	100	100
Spearman's ρ between package size and coverage				−0.33	−0.58

For Android (see Fig. 5 and Table 21, column "Coverage/How-to-do-it"), we observed that nine packages (out of 66) are completely covered, but they are also small (maximum ten elements). The second largest package with 166 elements, `android.widget`, is reasonably well covered (90.96%). This package contains (mostly visual) UI elements. However, the largest package with 169 elements (`android.provider`) is only half covered (50.89%). This package provides convenience classes to access the content providers that store common data, such as contact information, calendar information, and media files. There are also three smaller Android packages completely uncovered (scratched rectangles): `android.drm`, `android.mtp` and `android.service.textservice`. We observed that there is a very weak positive correlation between coverage and package size, $\rho = 0.04$. It suggests that there is no tendency for coverage increases/decreases when package size increases.

So far, we have presented results of the coverage analysis considering if API elements are covered or not. In addition, we present, for the covered API elements, the distribution of the numbers of threads (containing *how-to-do-it* question) to which these were linked. Figures 6a and 7a present the distribution for Swing and Android elements, respectively, in boxplots. Note that the y-axis is in a log scale.

Highlighted here is that, in general, the distributions for both Swing and Android follow a similar pattern. The long-tailed distribution indicates that only a small part of the API is responsible for the largest numbers of threads. For Swing, 50% of the covered elements are covered by numbers of threads in the interval [1, 8], and for Android in the interval [1, 14], i.e., a large part of the covered elements are not related to a very large number of threads.

Also, interesting outliers are noted in those boxplots indicating that a very small part of the APIs has a very high number of threads (hundreds, even thousands in the extreme cases). For Swing, any number of threads in the interval [95, 3645] is considered outlier, and for Android, in the interval [169, 16124].

RQ #1. To what extent does Stack Overflow cover API elements of Swing and Android APIs with threads containing *how-to-do-it* question?

Swing is covered on around 45% of all elements and Android is covered on around 69%. However, if we consider only top-level elements, Swing coverage is around 66% and Android coverage is around 77%. A few of small packages are completely covered and completely uncovered in both APIs. The distributions of the numbers of threads that were linked with API elements are long-tailed distributions for both APIs with significant outliers indicating that a small part of the APIs is responsible for the largest numbers of threads containing *how-to-do-it* question.

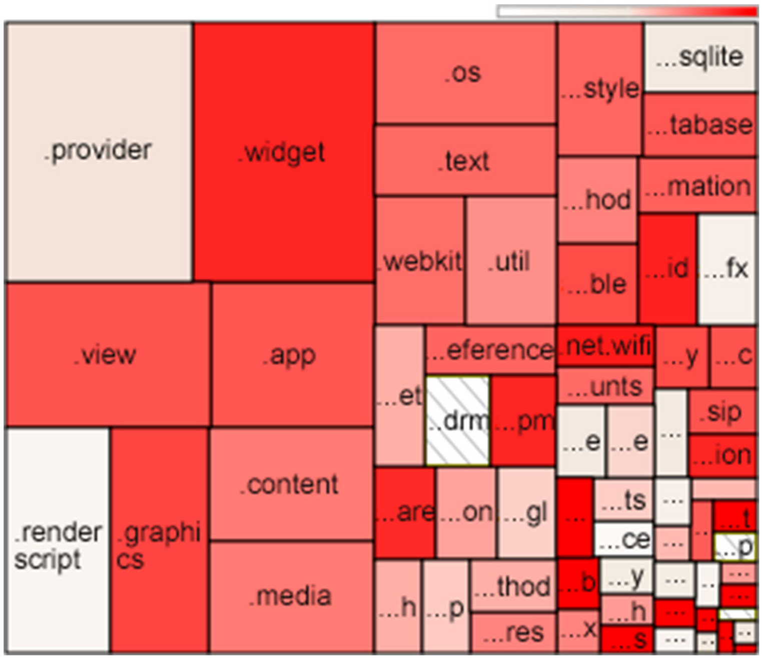


Fig. 5 Android intra-package coverage for *how-to-do-it* tasks considering the three API element types

Table 21 Android intra-package coverage considering the three API element types

		# Threads linked		Coverage (%)	
Package	# Elements	*How-to-do-it*	*Debug-corrective*	*How-to-do-it*	*Debug-corrective*
.provider	169	2580	6448	50.89	66.27
.widget	166	29,091	87,204	90.96	96.99
.view	104	21,371	66,212	81.73	90.38
.app	85	13,630	65,509	81.18	90.59
.renderscript	84	482	2235	19.05	57.14
.graphics	78	8,383	19,927	84.62	91.03
.content	68	23,222	63,025	73.53	92.65
.media	68	1432	5671	73.53	82.35
.os	64	16,144	74,437	76.56	93.75
.text	44	2287	4843	77.27	88.64
.util	42	9773	37,691	69.05	85.71
.webkit	42	1,488	5,095	76.19	88.1
.text.style	41	401	343	78.05	78.05
.database.sqlite	28	801	6554	46.43	89.29
.database	26	2,815	11,270	80.77	92.31
.text.method	25	250	644	72	72
android	24	23,318	75,180	91.67	87.5
.graphics.drawable	24	1960	5034	83.33	100
.media.audiofx	24	13	61	33.33	54.17
.net	24	4778	11,635	62.5	87.5
.view.animation	24	960	1882	79.17	87.5
.preference	23	1056	2675	78.26	82.61
.content.pm	22	1293	9834	90.91	86.36
.drm	22	0	6	0	9.09
.animation	20	135	310	65	80
.hardware	20	473	1933	90	90
.opengl	20	142	1,167	55	80
.bluetooth	16	216	850	62.5	75
.net.wifi.p2p	16	10	35	56.25	68.75
.view.inputmethod	16	530	1,263	62.5	75
.content.res	14	1761	5534	71.43	92.86
.net.wifi	14	313	559	92.86	100
.accounts	13	178	472	76.92	100
.gesture	13	38	129	53.85	84.62
.inputmethodservice	13	67	156	46.15	76.92
.telephony	12	605	1200	83.33	100
.app.backup	11	11	120	45.45	81.82
.location	11	1132	4726	90.91	100
.net.sip	11	8	48	81.82	90.91
.nfc	11	127	461	81.82	90.91
.nfc.tech	10	31	123	100	100
.speech.tts	9	79	375	55.56	88.89
.hardware.usb	8	19	129	100	100

Table 21 Android intra-package coverage considering the three API element types *(Continued)*

		# Threads linked		Coverage (%)	
Package	# Elements	*How-to-do-it*	*Debug-corrective*	*How-to-do-it*	*Debug-corrective*
.view.textservice	8	1	4	12.5	50
.sax	7	9	64	71.43	100
.view.accessibility	7	35	72	42.86	100
.graphics.drawable.shapes	6	57	164	100	100
.net.wifi.p2p.nsd	6	1	5	33.33	100
.speech	6	67	225	66.67	83.33
.appwidget	5	189	838	100	100
.mtp	5	0	1	0	20
.net.http	5	44	247	80	100
.net.nsd	5	1	6	40	100
.telephony.gsm	5	31	90	60	60
.text.util	5	59	115	60	60
.media.effect	4	3	17	25	75
.net.rtp	4	1	17	25	100
.text.format	4	513	1,473	100	100
.app.admin	3	34	90	66.67	100
.security	3	3	15	100	100
.accessibilityservice	2	12	33	100	100
.hardware.input	2	3	23	50	100
.os.storage	2	7	17	100	100
.service.textservice	2	0	5	0	100
.service.wallpaper	2	4	115	50	100
.telephony.cdma	1	3	9	100	100
Spearman's ρ between package size and coverage				0.04	−0.43

***Debug-corrective* coverage of API elements**

To answer RQ #2, we performed coverage analysis on the threads containing questions classified as *debug-corrective*: 17,547 threads related to Swing and 180,695 threads related to Android. Table 22 presents the coverage results. Noteworthy here is that a large number of the threads of both APIs was linked with some API element: 94.76% of the Swing threads (compared to 47.77% in *how-to-do-it* coverage) and 84.21% of the Android threads (compared to 46.28% in *how-to-do-it* coverage).

For Swing, we observed that there is at least one thread containing a *debug-corrective* question for 51.52% of the classes, 85.56% of the interfaces, and 80% of the enumerations, totaling 55.15% of the Swing elements. The coverage values of interfaces and enumerations are reasonably high, but just slightly more than half of the Swing classes are covered by the crowd regarding the purpose of documenting on how to solve common problems in the usage of API elements.

Moreover, we observed that 20.17% of the Swing inner classes are covered (slightly higher compared to 14.29% of *how-to-do-it* coverage). Considering only top classes, the coverage is higher (75.54%). Only 25% of the inner interfaces are covered, as observed for *how-to-do-it*. For enumerations, the two uncovered ones are inner enumerations.

For Android, we observed that there is at least one thread containing a *debug-corrective* question for 86.18% of the classes, 75.62% of the interfaces, and 75.86% of the enumerations, totaling 83.55% of the Android elements. These coverage values indicate that classes are better covered than interfaces and enumerations in Android, as the values of *how-to-do-it* coverage.

Regarding the inner elements, 75% of the Android inner classes are covered (compared to 58.52% in *how-to-do-it* coverage). Considering only top classes for Android, we observed that the coverage is reasonably high (90.52%), but it is not much higher than the overall class coverage. In regards to interfaces, 69.52% of the Android inner ones

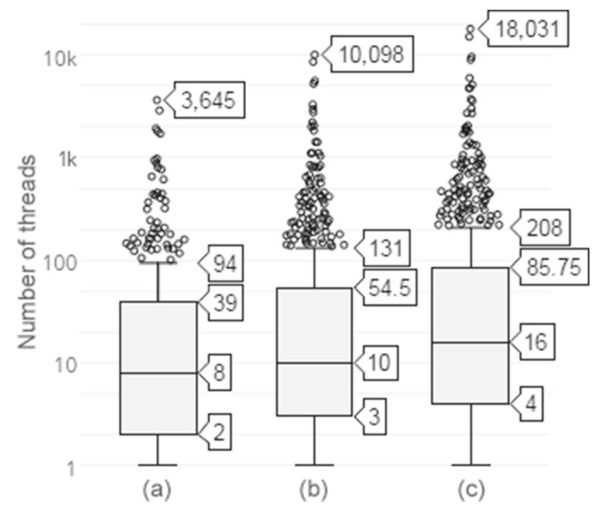


Fig. 6 Distribution of number of threads per Swing API element. **a** Coverage on threads with *how-to-do-it* question. **b** Coverage on threads with *debug-corrective* question. **c** Coverage on all API-related threads (replication of Parnin et al.'s work)

are covered (compared to 58.74% in *how-to-do-it* coverage). For enumerations, all the uncovered ones are inner enumerations.

We also analyzed the coverage of elements intra-package. The results for Swing are presented with different perspectives in Fig. 8 and Table 20, column "Coverage/Debug-corrective". We observed that three packages (out of 18) are completely covered: the two completely covered in RQ #1, `javax.swing.filechooser` and `javax.swing.text.rtf`, which

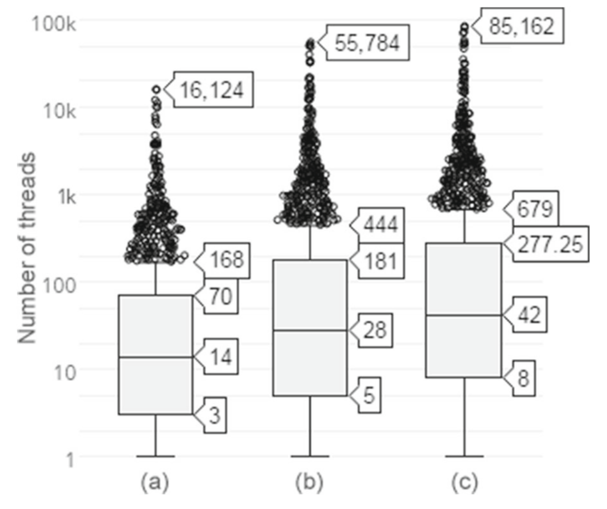


Fig. 7 Distribution of number of threads per Android API element. **a** Coverage on threads with *how-to-do-it* question. **b** Coverage on threads with *debug-corrective* question. **c** Coverage on all API-related threads (replication of Parnin et al.'s work)

are also the smallest ones, and the `javax.swing.border` package, the seventh smallest. Considering the two largest packages, *javax.swing* has an average coverage (68.05%), slightly better than in *how-to-do-it* coverage (63.07%), and `javax.swing.plaf.basic` is poorly covered (27.98%), although it is better than the *how-to-do-it* coverage (18.13%). Moreover, there is no package completely uncovered. We observed that only 3.23% of the elements of the package completely uncovered in RQ #1, `javax.swing.plaf.multi`, are covered in threads for *debug-corrective* tasks. We also observed that there is a moderate negative correlation between coverage and package size, $\rho = -0.58$, compared to a weak negative correlation in *how-to-do-it* coverage ($\rho = -0.33$).

For Android (see Fig. 9 and Table 21, column "Coverage/Debug-corrective"), we observed that 24 packages (out of 66) are completely covered, compared to nine packages in RQ #1. The second largest package (166 elements), `android.widget`, is well covered (96.99%). However, the largest package (169 elements), `android.provider` is not so well covered (66.27%), but still better covered compared to *how-to-do-it* coverage (50.89%). There is no package completely uncovered as in *how-to-do-it*. We also observed that there is a moderate negative correlation between coverage and package size, $\rho = -0.43$, compared to a very weak positive correlation in *how-to-do-it* coverage ($\rho = 0.04$).

In addition, we present, for the covered API elements, the distribution of the numbers of threads (containing *debug-corrective* question) to which these were linked. Figures 6b and 7b present the distribution for Swing and Android elements, respectively, in boxplots (y-axis is in a log scale).

Highlighted here is that, in general, the distributions for both Swing and Android follow a similar pattern. The long-tailed distribution indicates that only a small part of the API is responsible for the largest numbers of threads. For Swing, 50% of the covered elements are covered by numbers of threads in the interval [1, 10], and for Android in the interval [1, 28], i.e., a large part of the covered elements are not related to a very large number of threads.

Comparing to the distributions for threads from *how-to-do-it* questions, we observed that the median is higher: for Swing, the median is 8 for threads from *how-to-do-it* questions and 10 for threads from *debug-corrective* questions, and for Android, the median increased from 14 to 28.

Also, interesting outliers are noted in those boxplots indicating that a very small part of the APIs has a very high number of threads. For Swing, any number of threads in the interval [132, 10098] is considered outlier, and for Android, in the interval [445, 55784]. Compared to the distributions for threads from *how-to-do-it* questions, we

Table 22 Coverage results on Swing- and Android-related threads containing *debug-corrective* question type

API	# Threads	# Threads	Coverage considering all elements				Coverage considering only non-inner elements			
	analyzed	linked	Classes	Interfaces	Enums	Total	Classes	Interfaces	Enums	Total
Swing	17,547	16,628	51.52%	85.56%	80%	55.15%	75.54%	94.87%	100%	78.39%
			424	77	8	509	352	74	2	428
Android	180,695	152,166	86.18%	75.62%	75.86%	83.55%	90.52%	93.48%	100%	90.81%
			1,085	273	44	1,402	821	86	2	909

also observed that the maximum outlier increased from 3,645 to 10,098 for Swing and from 16,124 to 55,784 for Android.

RQ #2. To what extent does Stack Overflow cover API elements of Swing and Android APIs with threads containing *debug-corrective* question?

Swing is covered on around 55% of all elements and Android is covered on around 84%. However, if we consider only top-level elements, Swing coverage is around 78% and Android coverage is around 91%. Compared to coverage for *how-to-do-it* tasks, for both APIs, the coverage for *debug-corrective* tasks is higher. There is no package completely uncovered in both APIs, and more packages are completely covered compared to coverage for *how-to-do-it* tasks. The distributions of the numbers of threads that were linked with API elements are also long-tailed distributions for both APIs with significant outliers indicating that a small part of the APIs is responsible for the largest numbers of threads containing *debug-corrective* question.

Threads covering multiple API elements and their co-occurrence

To answer RQ #3, we calculate, for each API and for each coverage analysis (*how-to-do-it* and *debug-corrective*), how many threads linked with some API element cover multiple API elements. Table 23 presents the results as percentage values.

We observe that threads frequently cover multiple API elements. At least about 70% of the threads were linked with more than one API element. For coverage on threads containing *debug-corrective* question, almost 90% of the threads cover multiple API elements for both APIs.

In Fig. 10, we present, for the threads that cover multiple API elements, the distribution of number of API elements to which they were linked. Figure 10a, b presents the distributions for Swing coverage regarding threads with

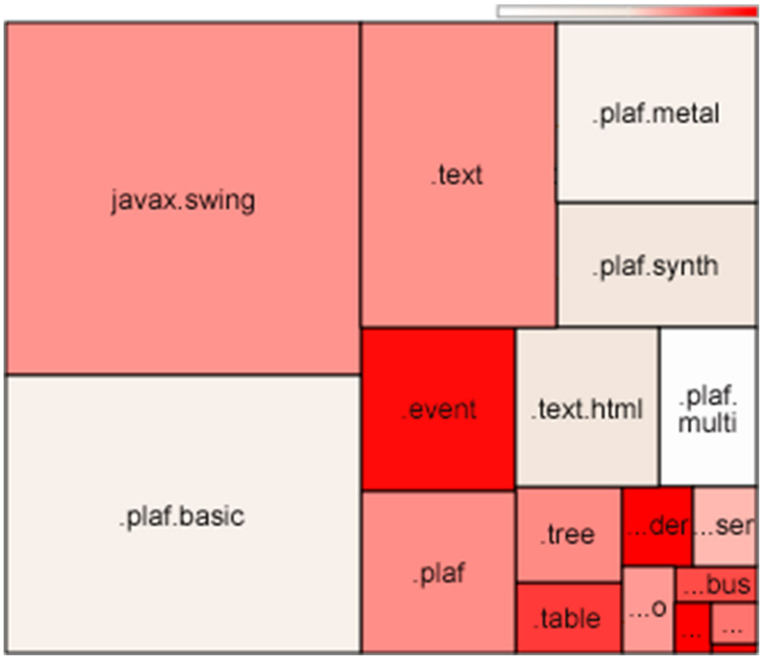


Fig. 8 Swing intra-package coverage for *debug-corrective* tasks considering the three API element types

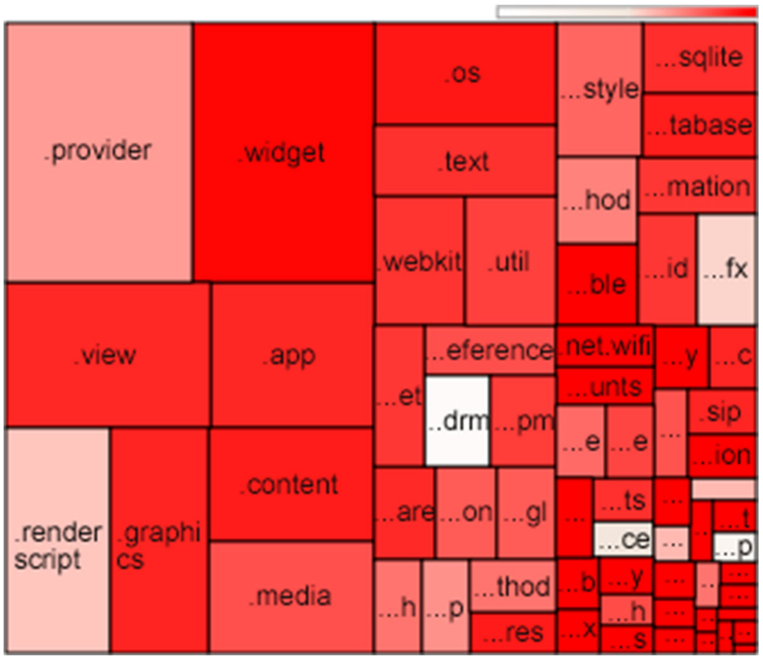


Fig. 9 Android intra-package coverage for *debug-corrective* tasks considering the three API element types

how-to-do-it and *debug-corrective* question, respectively, and Fig. 10c, d presents the distributions for Android coverage.

We noted that 50% of the threads cover a very small number of API elements—the higher median is six (Fig. 10d). Also, the outliers indicate that a small part of the threads was linked to a high number of API elements. For Swing, any number of API elements higher than 12 (*how-to-do-it*) and 13 (*debug-corrective*) is considered outlier, and for Android, any number higher than 12 (*how-to-do-it*) and 21 (*debug-corrective*). Based on those distributions, we can conclude that, in general, Android-related threads cover more API elements than Swing-related threads. In the extreme case, an Android-related thread covers 57 API elements.

We also analyzed the co-occurrence of API elements. To accomplish this, we used the Apriori algorithm for mining association rules between API elements. An association rule is an implication of the form $A \rightarrow B$, where A and B are sets of API elements. The idea of this rule is that threads covering the elements belonging to A *tend* to also cover the elements belonging to B.

Association rules are mined from a *database* containing a set of *transactions*, and each transaction has a set of *items*. In our case, threads are transactions, and API elements mentioned in the threads are itemsets.

We used R [43] for mining association rules by using the Apriori algorithm and for visualizing the results. Also, we had to define thresholds for *support* (*sup*) and *confidence*

Table 23 Percentage values concerning threads that cover multiple API elements for Swing and Android from both coverage analysis (*how-to-do-it* and *debug-corrective*)

API	*How-to-do-it*		*Debug-corrective*	
	# Threads linked	%Cover multi elem	# Threads linked	%Cover multi elem
Swing	8400	69.98	16,628	88.34
Android	70,325	73.23	152,166	87.27

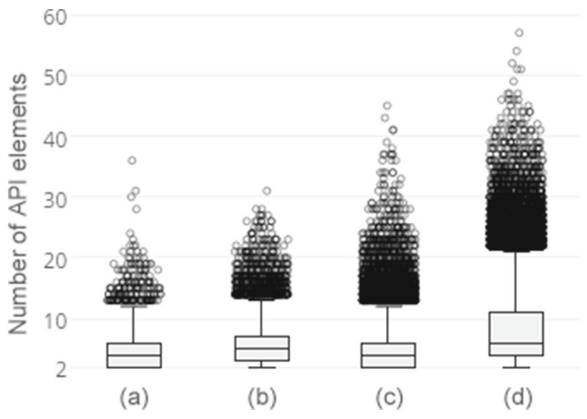


Fig. 10 Distribution of number of API elements per thread. **a**, **b** Swing coverage on threads with *how-to-do-it* and *debug-corrective* question, respectively; **c**, **d** Android coverage on threads with *how-to-do-it* and *debug-corrective* question, respectively

(*conf*) measures to find association rules of interest. The support of an itemset A is the proportion of transactions in the database which contain the itemset, and the confidence of a rule $A \rightarrow B$ is the proportion of transactions that support B among the transactions that support A.

The decision-making regarding the choice of thresholds for such measures is challenging. There is no consensus for such values. However, confidence of rules is more interesting than support of itemsets to us, i.e., to which degree can we rely on the occurrence of API elements implies that the occurrence of other API elements is more interesting than how much the API elements occur over the whole database.

Therefore, we choose higher thresholds for the confidence measure than for support. More specifically, we defined $sup = 0.01$ and we generated association rules for $conf = \{1.0, 0.9, \ldots, 0.5\}$. Tables 24 and 25 present the numbers of association rules obtained for Swing and for Android, respectively, from both *how-to-do-it* and *debug-corrective* coverage analysis.

Even the threshold for *sup* is low, the number of threads that API elements must be mentioned together for creating association rules is high. For instance, in the *how-to-do-it* coverage analysis for Swing, API elements must be mentioned together in at least 84 threads. It means that all association rules created have an absolute support that is reasonably high, and then we can claim that the API elements contained in the rules are frequently mentioned together.

We observe that, for both APIs, the numbers of association rules are higher for *debug-corrective* coverage analysis. This may suggest that threads containing *debug-corrective* question frequently mention more API elements that are frequently mentioned together than threads containing *how-to-do-it* question.

Table 24 Summary of the numbers of association rules obtained for Swing from both coverage analysis (*how-to-do-it* and *debug-corrective*)

	How-to-do-it	*Debug-corrective*
Threads	8400	16,628
Elements	418	509
Min sup	84	166
Association rules:		
$conf = 1.0$	5	79
$conf = 0.9$	106	582
$conf = 0.8$	224	1080
$conf = 0.7$	318	1430
$conf = 0.6$	417	1771
$conf = 0.5$	530	2102

Table 25 Summary of the numbers of association rules obtained for Android from both coverage analysis (*how-to-do-it* and *debug-corrective*)

	How-to-do-it	*Debug-corrective*
Threads	70,325	152,166
Elements	1162	1402
Min sup	703	1521
Association rules:		
$conf = 1.0$	0	47
$conf = 0.9$	261	8882
$conf = 0.8$	420	13,224
$conf = 0.7$	597	16,043
$conf = 0.6$	755	17,873
$conf = 0.5$	865	20,249

Regarding the association rules themselves, for presentation reasons, we do not present all of them. We choose the top seven rules based on confidence for each API, being that the top seven rules for Swing were obtained from threads containing *how-to-do-it* question, and the top seven rules for Android were obtained from threads containing *debug-corrective*.

Figures 11 and 12 present graphs for the seven association rules of the Swing and Android, respectively. API elements are represented by green nodes and implications of association rules are represented by orange nodes. The graph is interpreted by analyzing the directed edges—the incoming edges for an implication of association rule represent the antecedent API elements of the rule and the outcoming edges represent the consequent API elements of the rule.

For instance, by analyzing the implication in the top-left corner of the Fig. 11, we observe that there are two antecedent API elements, `javax.swing.JFrame` and `javax.swing.table.TableCellRenderer`, and there is one consequent API element, `javax.swing.JTable`. Then, the association rule is {`.JFrame`, `.table.Table- CellRenderer`} $\rightarrow$ {`.JTable`}, which means that when the `.JFrame` and `.table.TableCellRenderer` elements are mentioned in a thread, the `.JTable` element is also mentioned, with a confidence of 1.0, i.e., in all threads. Similarly, for Android (Fig. 12), the implication at the top-center refers to the association rule {`.content.pm.ComponentInfo`, `.content.Intent`, `.os.Lo- oper`, `.app.Instrumentation`} $\rightarrow$ {`.os.Handler`}.

These association rules are strong. Even the association rules generated with $0.5 \leq conf < 1.0$ are strong, since the absolute minimum support is reasonably high. By "strong", we mean that the API elements belonging to

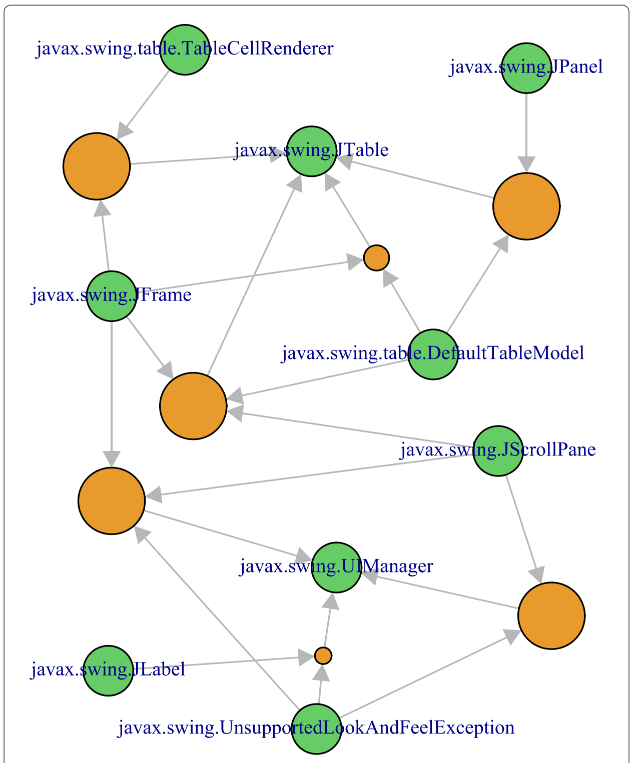


Fig. 11 Top seven association rules based on confidence for Swing obtained from threads containing *how-to-do-it* question. *Green nodes* represent API elements and *orange nodes* represent implications of association rules. The size of each *orange node* is proportional to the confidence of the rule

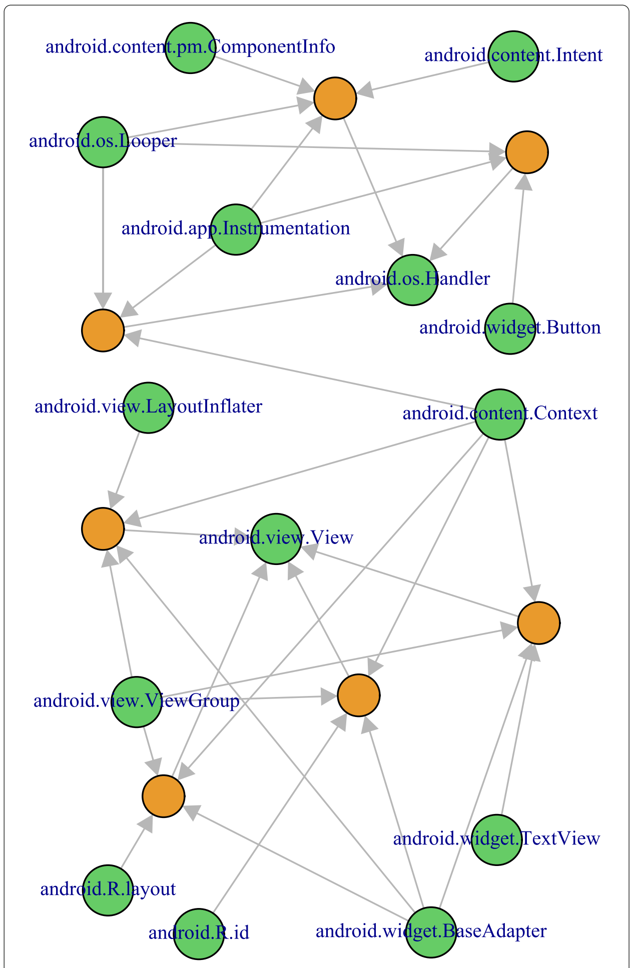


Fig. 12 Top seven association rules based on confidence for Android obtained from threads containing *debug-corrective* question

the generated association rules are always or frequently mentioned together.

We analyzed those most prominent co-occurrences to determine the reason why those API elements are always or frequently mentioned together. For Swing, we observed that there are two consequent API elements in the top-7 association rules: `.JTable` and `.UIManager`. The rules involving `.JTable` are:

- {`.table.TableCellRenderer`, `.JFrame`} → {`.JTable`}
- {`.table.DefaultTableModel`, `.JPanel`} → {`.JTable`}
- {`.table.DefaultTableModel`, `.JFrame`} → {`.JTable`}
- {`.table.DefaultTableModel`, `.JFrame`, `.JScrollPane`} → {`.JTable`}

`JTable` is used to display and edit regular two-dimensional tables of cells. It is understandable that `JTable` is mentioned together with `.table.TableCell- Renderer` and `.table.DefaultTableModel`, as these two elements are part of the `javax.swing.table` package, which provides classes and interfaces for dealing with `JTable`. `TableCellRenderer` is an interface that defines the method required by any object that would like to be a renderer for cells in a `JTable`. `DefaultTableModel` is an implementation of `TableModel`, an interface that specifies the methods the `JTable` will use to interrogate a tabular data model, that uses a Vector of Vectors to store the cell value objects. So we concluded that `JTable` is frequently mentioned together with `TableCellRenderer` and `DefaultTableModel` because of API design.

Regarding the involvement of `JFrame` in the rules, this class is a top-level Swing container, specialized to provide a place for other Swing components to paint themselves. `JTable`, `JPanel` (a generic lightweight container) and `JScrollPane` (a scrollable view of a lightweight component), which also participate in the rules, are examples of Swing components that always need to be put inside a `JFrame`. Moreover, `JPanel` and `JScrollPane` are mentioned with `JTable` because `JTables` are typically placed inside these components.

The association rules involving `.UIManager` are:

- {.UnsupportedLookAndFeelException, .JScrollPane, .JFrame} → {.UIManager}
- {.UnsupportedLookAndFeelException, .JScrollPane} → {.UIManager}
- {.UnsupportedLookAndFeelException, .JLabel} → {.UIManager}

UIManager manages the look and feel of GUI applications—look refers to the appearance of GUI widgets (as JComponents) and feel refers to the way the widgets behave. Due to the API design, the UnsupportedLookAndFeelException class is frequently mentioned together with UIManager, since it is an exception that indicates the requested look and feel management classes are not present on the user system. Moreover, JScrollPane and JLabel elements, as these are extensions of JComponent, it is natural that they are mentioned together with UIManager. However, it is a little bit arbitrary—it could be that the JButton element belongs to the association rules, for example, as any other extension of JComponent. Finally, since JComponents are used inside JFrames, it is not a surprise that the JFrame element also participates in one rule.

With Android, we also observe two consequent API elements in the top seven association rules: .os.Handler and .view.View. The rules involving .os.Handler are:

- {.os.Looper, .app.Instrumentation, .content.Intent, .content.pm.ComponentInfo} → {.os.Handler}
- {.os.Looper, .app.Instrumentation, .widget.Button} → {.os.Handler}
- {.os.Looper, .app.Instrumentation, .content.Conte- xt} → {.os.Handler}

Handler class allows to send and process Message and Runnable objects associated with a MessageQueue from a particular thread. Each Handler instance is associated with a single thread and the message queue for that thread. For API design reasons, Handler interacts with Looper (co-occurring in all three rules involving Handler), as it is used to run a message loop for a thread.

Instrumentation, that also is part of all those three rules, is a base class for implementing application instrumentation code. When running with instrumentation turned on, this class is instantiated before any of the application code, allowing to monitor all the interaction the system has with the application. In other words, it can be used to test activities as touching, clicks, typing, and other user actions relevant for Android applications.

An Intent is an abstract description of an action to be performed. Therefore, once a developer needs to monitor operations, he ends up with a class that holds instances of Instrumentation and Intent or even an Instrumentation class that has an Intent instance. A general use case that illustrates such co-occurrence is on developing Android application unit tests.

Context and ComponentInfo elements are not exactly related to the other API elements involved in the respective association rules with which they are co-occur. ComponentInfo is a base class containing information common to all application components, i.e., it shares common definitions between all application components. Context element is an interface to global information regarding an application environment, and it allows access to application-specific resources and classes, as well as up-calls for application-level operations such as launching activities and receiving intents. So that two API elements hold no semantic relationship with other API elements.

The association rules involving .view.View are:

- {.content.Context, .widget.BaseAdapter, .view.Vi- ewGroup, .view.LayoutInflater} → {.view.View}
- {.content.Context, .widget.BaseAdapter, .view.Vi- ewGroup, .widget.TextView} → {.view.View}
- {.content.Context, .widget.BaseAdapter, .view.Vi- ewGroup, .R.id} → {.view.View}
- {.content.Context, .widget.BaseAdapter, .view.Vi- ewGroup, .R.layout} → {.view.View}

View is the base class for widgets, which are used to create interactive user interface components. A View occupies a rectangular area on the screen and is responsible for drawing and event handling. It co-occurs with ViewGroup and TextView elements as they are direct subclasses of View. ViewGroup is the base class for layouts, which are invisible containers that hold other Views/ViewGroups and define their layout properties, and TextView displays text to the user and optionally allows them to edit it.

BaseAdapter is a common base class of common implementation for an Adapter that can be used in ListView, a view that shows items in a vertically scrolling list. BaseAdapter co-occur with View and ViewGroup as ListView is a non-direct subclass of View. LayoutInflater also co-occurs with View because it creates View objects based on layouts defined in XML.

Finally, after we analyzed the most prominent co-occurrences, we concluded that the main reason for the co-occurrence of API elements involved in strong association rules is due to API design. It may mean that the combined use of some API elements are potentially required for the use of other API elements that they are highly mentioned with. Consequently, this indicates that these elements should be documented together. Moreover, there are API elements involved in association rules that are not semantically related to the other API elements in the same rule. It means that the content concerning these API elements needs to be preprocessed for their documentation.

RQ #3. How often do threads cover multiple API elements?
Threads frequently cover multiple API elements (at least about 70% of them). Moreover, there are API elements that are always or frequently mentioned together. This may impose warnings concerning the use of the thread content for documenting these API elements, as the need of documenting API elements together or preprocessing the thread content to filter relevant content for specific API elements.

API coverage growth and API-related thread growth

To answer RQ #4, during coverage analysis, we obtained for each API element the date in which the element was covered (mentioned for the first time). Then, we calculated the coverage percentage of the APIs day by day, resulting in the growth of API coverage.

Moreover, we collected, from our Stack Overflow database, the number of threads related to the APIs (tagged with the API name) grouped by day. From the total number of threads related to each API, we calculated the percentage of threads day by day, resulting in the growth of API-related threads.

Figures 13 and 14 present charts, for Swing and Android, respectively, containing three series: the growth of API coverage on threads containing *how-to-do-it* (dotted line) and *debug-corrective* questions (dashed line), and the growth of the number of threads related to the API (solid line).

Noteworthy here are some observations based on the charts. First, in a general way, the growth of the two types of coverage follow a linear pattern on average, mainly for Swing. The growth of the number of threads related to the APIs, however, follow an exponential pattern.

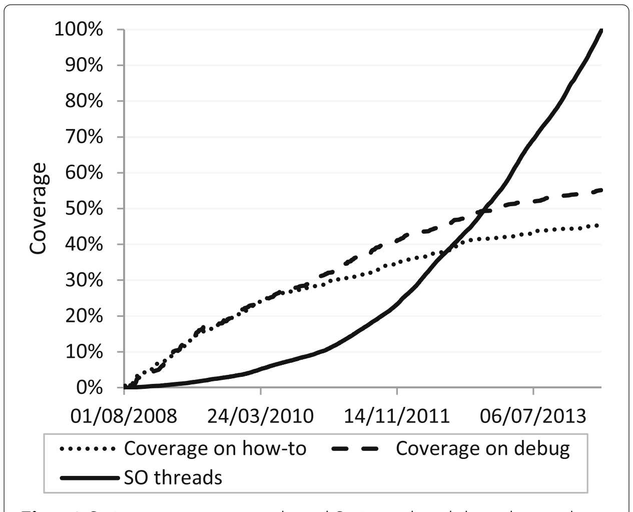


Fig. 13 Swing coverage growth and Swing-related thread growth

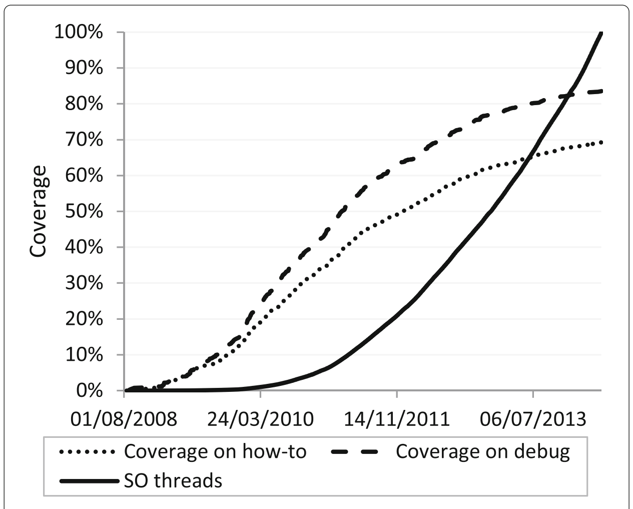


Fig. 14 Android coverage growth and Android-related thread growth

These patterns indicate that a high number of API elements were covered over a large time interval by a lower number of threads. For instance, from August 2008 until around June 2012, the percentage of coverage of the Swing elements was higher than the percentage of existing Swing threads at that time. For Android, the time interval is even larger.

Moreover, after these large periods, the numbers of threads related to the APIs continue to grow, overtaking the coverage, which continues to stabilize in a certain level. This indicates that Stack Overflow is reaching saturation at covering API elements after approximately four years for Swing and five years for Android.

In addition, we observed that, for each API, the coverage on threads containing *debug-corrective* questions is higher than the coverage on threads containing *how-to-do-it* questions over almost the whole period. This is consistent with the fact that the overall coverage of API elements by threads containing *debug-corrective* questions is higher.

RQ #4. How does Stack Overflow cover API elements over time?
Stack Overflow covers a high number of API elements with low number of threads in the first years. Then, Stack Overflow starts to reach saturation for covering API elements. This happens when the number of API-related threads still continues to increase, overtaking the growth of API coverage, which continues to stabilize. This indicates that API elements not covered have a high chance of never being covered, or take years to be. Consequently, the overall coverage of the APIs will not increase considerably

at the point of changing the fact that there are still many API elements not covered, so the content available on Stack Overflow may not be a complete substitute for official documentation of the APIs.

Coverage and actual use of APIs

To answer RQ #5, we calculate the Spearman's rank correlation coefficient between the number of threads which cover each API element (API coverage) and the number of projects which use each API element (API usage) obtained from Boa infrastructure [42]. We found usage for 635 (out of 923) Swing elements and usage for 1412 (out of 1678) Android elements.

Table 26 presents the correlation between API coverage and API usage for the two coverage analyses. We note that the correlations for Swing (both 0.805) are higher than the ones for Android (0.607 and 0.641), but for both APIs and for both coverage analyses, they are strong positive correlations (very strong for Swing). It means that API elements covered by a high number of threads are used in a high number of projects, and API elements covered by a low number of threads, or not covered, are used in a low number of projects, or not used.

Regarding those four packages completely uncovered in threads for *how-to-do-it* tasks, `javax.swing.plaf.multi`, `android.drm`, `android.mtp` and `android.service. textservice`, we observed that at least one element of each package is used in projects.

RQ #5. Is there an association between coverage of API elements on *how-to-do-it* and *debug-corrective* discussions by the crowd and actual usage of these elements in software systems?

There is a high association between coverage on Stack Overflow and actual use of API elements in software projects. This association seems to be largely influenced by the API, and less or not influenced by the type of question. It may suggest that widely used parts of an API require many documentation details that the crowd can address much more easily than the official documentation.

Comparing *how-to-do-it* and *debug-corrective* coverages with Parnin et al.'s coverage (all-inclusive)

Parnin et al. [8] reported in their study that 87.2, 77.3, and 54.3% of the Android, Java, and GWT classes, respectively, are covered by the crowd in the Stack Overflow.

Table 26 Spearman's rank correlation coefficient on API coverage and API usage

API	Coverage on *how-to-do-it*	Coverage on *debug-corrective*
Swing	0.805	0.805
Android	0.607	0.641

To quantify the difference of coverage considering threads with specific question types (*how-to-do-it* and *debug-corrective*) regarding coverage considering all threads, we replicated the Parnin et al.'s study. Replication was necessary because our experimental setup is different, e.g., we used the release of January 2014 of the Stack Overflow public data dump, and they used the release of December 2011.

Table 27 presents the coverage results obtained with our methodology (a summary of those presented in RQ #1 and RQ #2) and the coverage results obtained by replicating Parnin et al.'s work. We observe here that the coverage is higher for both API and for all API elements by analyzing all threads. This result may impose further warnings on the use of crowd knowledge depending on the documentation intent.

Moreover, we observed that more packages are completely covered when no thread filtering was applied. Also, the boxplots in Figs. 6c and 7c show that the numbers of threads per API element is also consistently higher in the percentiles, including the outliers.

Threats to validity

Our work has three main threats. First, we analyzed only two APIs (Swing and Android), and both are based on the same language (Java). Thus, our results cannot be generalized to other APIs. However, we have shown that coverage analysis was API-dependent suggesting that each API requires its own analysis, even within the same language.

Second, the classifier of question is not completely accurate. To mitigate that, we have chosen different classification algorithms for each API to take advantage of the best accuracy. Whenever classifying questions related to other APIs, either a training set must be created for it, or the generic classifier can be used under the penalty of a still slightly lower accuracy.

Table 27 Coverage results on Swing- and Android-related threads containing *how-to-do-it* and *debug-corrective* question types, and on all threads

API	# Threads	Coverage (%)			
	analyzed	Classes	Interfaces	Enums	Total
For *how-to-do-it*					
Swing	17,583	40.83	83.33	70	45.29
Android	151,955	71.17	64.27	58.62	69.25
For *debug-corrective*					
Swing	17,547	51.52	85.56	80	55.15
Android	180,695	86.18	75.62	75.86	83.55
For all threads					
Swing	38,345	56.87	88.89	90	60.35
Android	424,614	90.55	82.83	79.31	88.5

Third, the identification of the API element name mentions in the content of threads is made by exact word matching. Thus, a given API element is counted as covered if its name is spelt correctly. On the one hand, we can miss potential mentions, but on the other hand, we do not count false mentions. Also, another potential threat refers to name resolution when more than one element in the API has the same short name, and none of their packages is mentioned in the thread. The Swing "`Element`" element, for example, was encountered within code samples of 44 threads. However, these mentions were neither counted for `javax.swing.text.html.parser.Element` class nor for `javax.swing.text.Element` interface, since our approach cannot handle with name ambiguity. Nevertheless, this interface was covered by 16 threads, where its package name was mentioned, but the class remained not covered.

Limitations
The main limitation of our work is that we do not take into account which API elements the Stack Overflow discussions are actually about (semantic analysis). Our coverage results are based on an analysis of *mentions* of API elements in the content of threads, which does not mean that the central point of the discussions in the threads are about these API elements.

Despite this, we conducted a study in order to quantify a potential decrease in our coverage results when the semantics of Stack Overflow posts is taken into account (study reported in the "Analysis of actual semantic links" subsection). Moreover, we have analyzed the co-occurrence of API elements in the threads, which gave us some insights as, for example, that some API elements are potentially required for other API elements, as they are frequently mentioned together. Consequently, this can indicate that they are contextual elements in the threads, and can be eliminate in a process of discovering the central element of the thread. On the other hand, this does not mean that there is no information for documenting these elements only because they are not the central point of threads.

Another limitation is that we do not analyze the quality and confidence of the content of threads that cover API elements. These factors need to be considered when using the content of threads for documenting API elements. We intend to propose a method for, given an API element, ranking the threads linked with it based on the quality and confidence of the content of the threads, aiming to select the best threads for documenting that API element. Information concerning the quality and confidence of the content of the threads can be extracted, for example, through existing mechanisms on Stack Overflow, as the up/down voting on questions and answers, which indicates whether they are useful.

Practical implications
The results of our study have practical implications, which may be of interest to central authorities and researchers who intend to (re)document APIs, and to software library/framework developers and API designers.

Regarding overall coverage of an API, poor coverage may suggest that it is not feasible to use the crowd knowledge for generating an API documentation as a substitute for official documentation. However, it is reasonable to claim that hardly an API will be well covered by the crowd (coverage $> 90\%$), since a part of the API elements are not being used in practice, and the API may have elements too simple for use. It may suggest that a part of the API does not need additional information besides that which already exists in the official documentation of the API.

The fact that there are API elements that are not discussed by the crowd and are not being used in software systems also may suggest that these elements are not useful for developers. Moreover, the strong association between API coverage and its usage may suggest that widely used elements have more chances of different nuances, and therefore they need to be carefully documented to support their different nuances. Elements not widely used, on the other hand, demand less different nuances, so they are discussed less by the crowd.

Furthermore, there are API elements that are always or frequently mentioned together in threads. In the most cases, it may mean that the combined use of some API elements are potentially required for the use of other API elements with which they are commonly mentioned. Consequently, this indicates that these elements should be documented together. In other cases, there are API elements involved in association rules that are not semantic related to the other API elements in the same rule. It means that the content concerning these API elements needs to be preprocessed for their documentation. This may impose warnings concerning the use of the thread content for documenting these API elements, as the need of documenting API elements together or preprocessing the thread content to filter relevant content for specific API elements.

Related work
(Re)documenting APIs
Due to the lack of sufficient examples and explanations in API documentation, when that documentation exists, a new research field has emerged, in which methods have been proposed to automatically (re)document APIs [5, 8, 44–51]. We refer to *documenting APIs* as the process of creating documentation for an API, and *redocumenting*

APIs as the process of creating improved documentation based on that which already exists.

For both processes, we can generally identify that documentation generation involves the definition of three: (1) *scope* and (2) *structure* of the documentation, and (3) *data source* for documentation generation.

Scope consists of the type(s) of content that should be included in documentation, which is defined by the *intentions* from the point of view of the API users, i.e., what they want to do with or to know about a given API. Some common intention types are, for example, how to use individual API elements (as in Javadoc-like documents), how to implement specific tasks using an API [5], and how to fix domain-independent bugs in an existing code where there was misunderstanding regarding the usage of an API (crowd-bugs) [52].

Thus, API documentation can be intention-oriented: it may include content for different intention types (general documentation) or different documentations may include content for specific intention types (specific documentations), where they are complementary to each other. As advantages, specific documentation can assist API users in their search for what they want based on their intention type, and can have different structures depending on their scope.

Structure consists of the organization and presentation of documentation. Some examples are Javadoc, cookbook and frequently asked questions (FAQs).

Javadoc is a well-known way for documenting Java code. A Javadoc document is created for API intermediate elements (class, interface and enumeration), which basically includes a general textual description of the element and also a description of its members (such as fields and methods). Javadoc usually does not include code examples [8]. Moreover, there is no organization of Javadoc to assist developers to find what they want in regards to the API and there is no explanation concerning the use of classes/methods together in order to perform a specific task.

Cookbooks, on the other hand, have semi-structured content based on chapters and recipes [5, 44, 45]. A cookbook is composed of chapters on a specific theme and each chapter is composed of recipes. Each recipe contains a programming task and instructions on how to accomplish that task by using elements of the API. Unlike Javadocs, which provide a textual description of each API element individually, cookbooks provide solutions, including, besides textual explanation, source code examples possibly involving more than one API element to accomplish tasks.

Differently, FAQ-like documentation consists of a structure which lists questions frequently asked by the target audience and the corresponding expert answers [46]. This type of structure is not used to document a specific API element or functionality. Actually, the purpose of FAQs is to provide independent pieces of knowledge targeting practical problems [46], therefore, they are usually complementary to other documentation.

The generation of documentation is performed by using some *data source* that contains relevant content for API documentation. In addition to the API source code itself, software repositories and social media, such as GitHub and Q&A sites, have recently been adopted as a documentation data source.

By using the API source code, some works automatically generate natural language summaries for API elements that can be used in the description of those elements in the documentation. Moreno et al. [47] proposed a technique to generate summaries for Java classes, in which their intentions are described briefly. McBurney and McMillan [48] proposed a technique to generate summaries for API methods instead of API classes, and Sridhara et al. [49] to generate comments for method parameters.

Other data sources used to (re)document APIs are those that may provide code examples of API usage. APIMiner tool [50] and the Kim et al.'s approach [51] automatically extract such code examples from (private and web, respectively) source code repository to enhance existing Javadocs.

Souza et al. [5] used the crowd knowledge available on Stack Overflow to semi-automatically generate cookbooks for API documentation. Generated cookbooks consist of the description of programming tasks and their solutions by using the API. Lafetá et al. [45] also generated cookbooks, focusing on framework feature instantiation instead of general programming tasks. They mined examples of instantiations from the source code of the framework itself and from existing instantiations to generate cookbooks.

Treude and Robillard [53] also used crowd knowledge available on Stack Overflow, as Souza et al. [5] did, to automatically augment API documentation with insight sentences. They defined that insight sentences are sentences related to a particular API type and that provide insight not contained in the API documentation of that type. Their idea is to summarize new documents assuming that the reader is already familiar with certain old documents.

Henβ et al. [46] proposed a semi-automated approach to extract FAQs from two software development support channels, forums and mailing lists, in order to document APIs. The FAQs are extracted by identifying popular topics, such as compiler errors, and are proposed to be published as official software documentation [46].

Our study sheds light on some limitations concerning coverage when using Stack Overflow to (re)document APIs. Moreover, we have shown that coverage results are different for different kinds of questions and thus can create different influence depending on the intent of the (re)documentation.

Stack Overflow

Stack Overflow has been studied in order to help researchers to understand the knowledge/mechanisms available on it and how these can be used to assist software development.

Nasehi et al. [9] defined that the types of question made on Stack Overflow can be described based on two dimensions: (1) the question topic, such as the main technology that the question involves and (2) the main concerns of the asker. They defined four question categories for the second dimension. Treude et al. [10] also defined categories on the main concerns of askers, and they found out that Stack Overflow is effective in code reviews (*review* question type), for *conceptual* questions and for *novice*, and the most frequent type of question is *how-to* and questions about unexpected behavior (*discrepancy*), which is consistent with our finding, reinforcing the decision to study those two kinds of question.

The automatic identification of question topics is usually carried out using question tags. On the automatic identification of question type involving the main concerns of the asker, Souza et al. [7] conducted an experimental study for the automated classification of Q&A pairs into three categories: *how-to-do-it*, *conceptual*, and *seeking-something*. They performed a comparison between different classification algorithms, and selected a logistic regression algorithm that had the best performance with an overall success rate of 76.19 and 79.81% on the *how-to-do-it* category.

In our most related work, Parnin et al. [8] conducted a study on the feasibility of using the crowd knowledge available on Stack Overflow for documenting three APIs: Java, Android and GWT. They found that 87.2, 77.3, and 54.3% of the Android, Java, and GWT classes, respectively, are covered by the crowd. The main difference between our work and the Parnin et al.'s work is that they analyzed coverage of API elements in a general way, with no criterion or filter on the Stack Overflow threads regarding the type of API documentation that they desired to target. Hence, an understanding of how the API elements are covered by the crowd and how the crowd knowledge can be used for generating API documentation was not possible. In our work, we have conducted a coverage analysis according to types of questions related to the main concerns of askers, and thus, subsidizing the choice of threads for different types of API documentation intent. We also presented in the "Comparing *how-to-do-it* and *debug-corrective* coverage with Parnin et al.'s coverage (all-inclusive)" section, the difference in coverage concerning our approach and the Parnin et al. approach by replicating their work.

Linking documents with code elements

There are two general approaches to link documents with code elements: (1) from a list of code elements, code elements are searched for in documents, and when an element is found, it is linked with the document [8, 13], and (2) from documents, code-like terms are identified and mapped to their corresponding code elements and then, the documents are linked with the code elements [37, 54]. A code-like term is a series of characters that matches a pattern associated with a code element kind (e.g., camel cases for types and parentheses for functions) [54].

Parnin et al. [8] and Linares-Vásquez et al. [13] identified links between Stack Overflow threads and API classes previously known. Their approaches are based on exact word matching of class names in different types of textual content (link types) on Stack Overflow posts: code sample, code markup, href markup, word, and title links. The main difference between the two works is how they handled collisions of class names with English words. To avoid false positives regarding this, Parnin et al. excluded word links for one-word API class names. Linares-Vásquez et al. pipelined the link-type detectors, and when a detector identifies a link, the next detector is not executed. The sequence of the detectors was thus organized in a way that code and href link detectors are executed before word and title link detectors, decreasing the possibility of false positives.

Dagenais and Robillard [54] developed a tool called RecoDoc that relies on a technique which identifies code-like terms in developer documentation and support channels and links these terms to fine-grained code elements in an API (e.g., class, method, field). They identify code-like terms from free-form text by using lightweight regular expressions and from code fragments by using partial program analysis (PPA). Their technique is based on the assumption that code elements mentioned in close vicinity are more likely to be related than code elements mentioned further apart, so they take into account the context in which a code-like term is mentioned—context refers to additional information in various scopes surrounding the term [37]. They also identified sources of ambiguity inherent to linking code-like terms in unstructured natural language documents. To resolve these ambiguities, they applied a set of filtering heuristics. For example, for a given term identified as a member (e.g., method, field), the declaration ambiguity filter analyzes the context of this term trying to find its declaring type.

Rigby and Robillard's tool, called ACE [37], is based on the technique implemented on RecoDoc. ACE, however, has two main improvements concerning linking code-like terms with code elements. First, they developed an island parser based on constructs in the Java Language Specification in order to identify code-like terms from free-form text and code fragments. Island parsers are generated from island grammars—grammars that consist of detailed productions that describe the language constructs of interest (the islands), and liberal productions that catch the remainder (the water) [55, 56]. Therefore, ACE is able to identify code-like terms from source code that did not compile, different to RecoDoc that uses PPA creating a dependence on the Eclipse Java compiler. Second, ACE does not rely on a list of code elements previously known as RecoDoc does. Instead, ACE identifies code elements and creates an index of valid elements based on the elements contained in the collection of documents. Then, ACE reparses each document extracting unqualified, ambiguous terms and resolves these for their corresponding code elements by using the context of the terms. Therefore, ACE is also able to identify code elements in informal documents from multiple APIs.

Subramanian et al. [57] proposed Baker, a tool for automatically generating links between type references, method calls, and field references identified in source code examples to API documentation. To solve ambiguities, they proposed a process called deductive linking. It generates an incomplete abstract syntax tree (AST) for the source code being analyzed, and uses information from an oracle (dictionary) for Java and Javascript to deduce facts about the AST. Baker performs this deduction iteratively, where in each iteration, the tool performs a depth-first traversal of the AST and examines all nodes of interest: nodes involved in declarations, invocations, and assignments. For each of these nodes, Baker builds a list containing the potential matches for that element from the oracle. The iteration continues until either all elements are associated with a single fully qualified name or an iteration fails to improve the results for any element.

Our linking implementation described in the “Linking Stack Overflow threads with API elements” section relies on the approaches of Parnin et al. [8] and Linares-Vásquez et al. [13]. We presented, in the section, an evaluation of our implementation and showed that precision was 99.20% and recall was 94.68%. We cannot compare the reliability of our implementation with that of Parnin et al. because they did not present an assessment for it. Linares-Vásquez et al. evaluated their implementation, but different to ours, they analyzed precision of links between Stack Overflow content and API methods, and as such it was impossible for us to make a fair comparison.

Conclusions

In this paper, we reported a study on the coverage of the Swing and Android elements on Stack Overflow for API documentation regarding *how-to-do-it* and *debug-corrective* tasks.

We found that, for both tasks, the overall coverage values for Android elements (around 69% for *how-to-do-it* and 84% for *debug-corrective*) are higher than the overall coverage values for Swing elements (45 and 55%). In addition, we observed that those coverage values increase if we do not consider inner elements: for Swing, at least 20% increases and for Android at least 7%. Therefore, we conclude that inner elements are poorly covered.

Concerning the two tasks, *debug-corrective* has better coverage than *how-to-do-it*, for each API, in four ways: (1) the overall coverage values are higher, (2) the API elements are covered by more threads, (3) all API packages have at least one element covered, and (4) the covering speed is higher.

Furthermore, there are API elements that are always or frequently mentioned together in threads. We observed that, in the most cases, these API elements are potentially required for the use of other elements. Then, these API elements require a careful selection of the threads content for their appropriate documentation.

Additionally, the analysis on the association between the coverage of APIs and their usage in real software systems revealed that these two variables are strongly associated. This may suggest that widely used elements have more chances of different nuances that take advantage of more specific code samples and explanations (documentation) tailored for that specific use. Elements not widely used demand less different nuances and consequently are less discussed by the crowd.

Finally, through a manual and qualitative study, we have raised an important issue on the nature of semantic links for Stack Overflow posts, and we have shown that the meaning of semantic relevance is not only different for *how-to-do-it* and *debug-corrective* posts (something that is even new in the related literature), but also that irrelevant elements in *debug-corrective* posts are much more frequent than in *how-to-do-it* posts. As further work, we intend to propose an automatic solution to link Stack Overflow posts with API elements at a semantic level.

One point for further investigation is whether the inner elements are important for API documentation. Since they are public, they are part of the API, and as so, should be documented. Also, it is worth investigating the reasons why there are still non-inner elements that are not covered, whether they are simple to use and do not require discussion by the crowd, or they are not widely used and so the chances of discussion are reduced.

Also, mechanisms could be proposed to motivate the crowd at covering the non-simple elements that are not covered, in order to increase the API coverage, and consequently to feasible the API documentation by the crowd. Moreover, taking advantage of the high coverage for *debug-corrective* tasks, a new kind of debugging assistant would be conceived to contextually match the source code of developers with the code in questions, and then to recommend the fix based on the code in the answers.

Appendix

Attribute selection based on the information gain method

Tables 28, 29, and 30 present information about the attribute selection based on the information gain method for the Android, Swing, and both training sets, respectively. In each table, the top five attributes with the highest information gain value for each one of the three settings of the tests (1B, 2B, and 3B) are presented, where attribute selection is performed. We note here that for each setting, the top-5 attributes are basically the same, and the QUESTION_CODE_SIZE attribute is the most discriminative attribute in general. Additionally, we present in the column "#Selected" the number of selected attributes out of the number of total attributes entered for each test setting.

Table 28 Top five attributes selected and their respective information gain value for the Android training set on the three settings for the tests

Setting	#Selected	Top five attributes	Info gain value
1B	10/12	DEBUG_KEYWORDS	0.353418158
		QUESTION_CODE_SIZE	0.350187704
		QUESTION_HAS_CODE	0.284593264
		QUESTION_BODY_SIZE	0.276840173
		HOW_KEYWORDS	0.202119286
2B	25/153	QUESTION_CODE_SIZE	0.350187704
		QUESTION_HAS_CODE	0.284593264
		QUESTION_BODY_SIZE	0.276840173
		howto_KEYWORD	0.265824527
		error_KEYWORD	0.137824814
3B	10/12	QUESTION_CODE_SIZE	0.350187704
		DEBUG_KEYWORDS	0.30618482
		HOW_KEYWORDS	0.294084392
		QUESTION_HAS_CODE	0.284593264
		QUESTION_BODY_SIZE	0.276840173

Table 29 Top five attributes selected and their respective information gain value for the Swing training set on the three settings for the tests

Setting	#Selected	Top five attributes	Info gain value
1B	9/12	QUESTION_CODE_SIZE	0.408890225
		DEBUG_KEYWORDS	0.338259094
		QUESTION_HAS_CODE	0.32382105
		QUESTION_BODY_SIZE	0.271777671
		HOW_KEYWORDS	0.170159934
2B	22/153	QUESTION_CODE_SIZE	0.408890225
		QUESTION_HAS_CODE	0.32382105
		QUESTION_BODY_SIZE	0.271777671
		howto_KEYWORD	0.157489004
		problem_KEYWORD	0.069082363
3B	9/12	QUESTION_CODE_SIZE	0.408890225
		QUESTION_HAS_CODE	0.32382105
		DEBUG_KEYWORDS	0.312655378
		QUESTION_BODY_SIZE	0.271777671
		HOW_KEYWORDS	0.174419188

Table 30 Top five attributes selected and their respective information gain value for both Android and Swing training sets on the three settings for the tests

Setting	#Selected	Top five attributes	Info gain value
1B	10/12	QUESTION_CODE_SIZE	0.377347749
		DEBUG_KEYWORDS	0.330994105
		QUESTION_HAS_CODE	0.305653075
		QUESTION_BODY_SIZE	0.27711226
		HOW_KEYWORDS	0.178311939
2B	32/153	QUESTION_CODE_SIZE	0.377347749
		QUESTION_HAS_CODE	0.305653075
		QUESTION_BODY_SIZE	0.27711226
		howto_KEYWORD	0.1949863
		error_KEYWORD	0.09552362
3B	10/12	QUESTION_CODE_SIZE	0.377347749
		DEBUG_KEYWORDS	0.321029275
		QUESTION_HAS_CODE	0.305653075
		QUESTION_BODY_SIZE	0.27711226
		HOW_KEYWORDS	0.194580371

Acknowledgements
FMD would like to thank Fabíola Souza Fernandes Pereira for reviewing some parts of the paper and for the discussions concerning the classifier.

Funding
We would like to thank FAPEMIG, CAPES, and CNPq for partially funding this work. We also gratefully acknowledge the financial support of CAPES, CSNR, and CONICYT under the STIC-AmSud project "Mining software repositories to instantiate software frameworks and react to API change".

Availability of data and materials
The datasets supporting the conclusions of this article are available at http://lascam.facom.ufu.br at Downloads.

Authors' contributions
FMD collected data, manually classified the training sets, performed the experiments, analyzed the results, prepared the figures and tables, and wrote the manuscript. KVRP collected data, manually classified the training sets, assisted the experimental executions, and worked on the manuscript. DC participated in the discussions between the authors regarding the work and worked on the manuscript. MAM conceived the main idea of the work, coordinated the work, and worked on the manuscript. All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Faculty of Computing, LASCAM-FACOM, Federal University of Uberlândia, Uberlândia, Brazil. [2]RMoD Inria Lille-Nord Europe, University of Lille-CRIStAL, Lille, France.

Received: 8 March 2016 Accepted: 2 November 2016
Published online: 09 December 2016

References

1. Robillard MP (2009) What Makes APIs Hard to learn? Answers from developers. IEEE Software 26(6):27–34
2. Robillard MP, DeLine R (2011) A field study of API learning obstacles. Empir Softw Eng 16(6):703–732
3. Parnin C, Treude C (2011) Measuring API documentation on the Web. In: Proceedings of the 2nd International Workshop on Web 2.0 for Software Engineering (Web2SE'11). ACM, New York, pp 25–30
4. Ponzanelli L, Bacchelli A, Lanza M (2013) Leveraging crowd knowledge for software comprehension and development. In: Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR' 13). IEEE Computer Society, Washington, DC, pp 57–66
5. Souza LBL, Campos EC, Maia MA (2014) On the extraction of cookbooks for APIs from the crowd knowledge. In: Proceedings of the Brazilian Symposium on Software Engineering (SBES'14). IEEE, Maceió, pp 21–30
6. Stack Exchange Inc (2015) Stack Overflow. http://stackoverflow.com/. Accessed 30 Nov 2016
7. Souza LBLd, Campos EC, Maia MDA (2014) Ranking crowd knowledge to assist software development. In: Proceedings of the 22nd International Conference on Program Comprehension (ICPC'14). ACM, New York, pp 72–82
8. Parnin C, Treude C, Grammel L, Storey MA (2012) Crowd documentation: exploring the coverage and the dynamics of API discussions on Stack Overflow. Technical Report GIT-CS-12-05. Georgia Institute of Technology. http://www.chrisparnin.me/pdf/crowddoc.pdf
9. Nasehi SM, Sillito J, Maurer F, Burns C (2012) What makes a good code example? A study of programming Q&A in StackOverflow. In: Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM'12). IEEE Computer Society, Washington, DC, pp 25–34
10. Treude C, Barzilay O, Storey MA (2011) How Do Programmers ask and answer questions on the Web? (NIER Track). In: Proceedings of the 33rd International Conference on Software Engineering (ICSE'11). ACM, New York, pp 804–807
11. Delfim FM, da Paixão KVR, de A Maia M (2015) Redocumenting APIs with crowd knowledge: a study on the coverage of the Swing API on Stack Overflow (in Portuguese). In: III Workshop on Software Visualization, Evolution, and Maintenance (VEM'15). Belo Horizonte, pp 1–8
12. Kavaler D, Posnett D, Gibler C, Chen H, Devanbu P, Filkov V (2013) Using and asking: APIs used in the Android market and asked about in StackOverflow. In: Proceedings of the 5th International Conference of Social Informatics (SocInfo' 13), Volume 8238 of the Series Lecture Notes in Computer Science. Springer International Publishing, Cham, pp 405–418
13. Linares-Vásquez M, Bavota G, Di Penta M, Oliveto R, Poshyvanyk D (2014) How do API changes trigger Stack Overflow discussions? A Study on the Android SDK. In: Proceedings of the 22nd International Conference on Program Comprehension (ICPC' 14). ACM, New York, pp 83–94
14. Campos E, Souza L, Maia M (2016) Searching Crowd Knowledge to Recommend Solutions for API Usage Tasks. Journal of Software: Evolution and Process (JSEP). Wiley, p 31
15. Campos EC, de Almeida Maia M (2014) Automatic categorization of questions from Q&A sites. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC' 14). ACM, New York, pp 641–643
16. Aha DW, Kibler D, Albert MK (1991) Instance-based learning algorithms. Mach Learn 6(1):37–66
17. Quinlan JR (1993) C4.5: Programs for machine learning. Morgan Kaufmann Publishers Inc., San Francisco
18. Quinlan JR (1996) Improved use of continuous attributes in C4.5. J Artif Intell Res 4(1):77–90
19. John GH, Langley P (1995) Estimating continuous distributions in Bayesian classifiers. In: Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI'95). Morgan Kaufmann Publishers Inc., San Francisco, pp 338–345
20. Friedman N, Geiger D, Goldszmidt M (1997) Bayesian network classifiers. Mach Learn Spec Issue Learn Probabilistic 29(2–3):131–163
21. Kohavi R (1995) The power of decision tables. In: Proceedings of the 8th European Conference on Machine Learning (ECML' 95). Springer, London, pp 174–189
22. Silva LM, Marques de Sá J, Alexandre LA (2008) Data classification with multilayer perceptrons using a generalized error function. Neural Netw 21(9):1302–1310
23. Platt JC (2002) Advances in kernel methods. pp. 185–208. MIT Press, Cambridge, MA, USA (1999) .Chap. Fast Training of Support Vector Machines using Sequential Minimal Optimization
24. Breiman L (2001) Random forests. Mach Learn 45(1):5–32
25. Landwehr N, Hall M, Frank E (2005) Logistic model trees. Mach Learn 59(1–2):161–205
26. le Cessie S, van Houwelingen JC (1992) Ridge estimators in logistic regression. Appl Stat 41(1):191–201
27. Nigam K, Lafferty J, McCallum A (1999) Using maximum entropy for text classification. In: IJCAI-99 Workshop on Machine Learning for Information Filtering. Stockholm, pp 61–67
28. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The WEKA Data Mining Software: an update. ACM SIGKDD Explorations Newsl 11(1):10–18
29. McCallum AK MALLET: A machine learning for language toolkit. http://mallet.cs.umass.edu. Accessed 30 Nov 2016
30. Landis JR, Koch GG (1977) The measurement of observer agreement for categorical data. Biometrics 33(1):159–174
31. Cunningham P (2000) Overfitting and diversity in classification ensembles based on feature selection. Technical Report TCD-CS-2000-07, Department of Computer Science, Trinity College Dublin, Dublin, Ireland
32. Yu L, Liu H (2004) Efficient feature selection via analysis of relevance and redundancy. J Mach Learn Res 5:1205–1224
33. Refaeilzadeh P, Tang L, Liu H (2009) Cross-validation. In: Liu L., Özsu M. T. (eds). Encyclopedia of database systems. Springer, New York, pp 532–538
34. McNemar Q (1947) Note on the sampling error of the difference between correlated proportions or percentages. Psychometrika 12(2):153–157
35. Bostanci B, Bostanci E (2013) An evaluation of classification algorithms using McNemar's Test (Bansal CJ, Singh KP, Deep K, Pant M, Nagar KA, eds.). Springer, India
36. (2016) Jericho HTML Parser. http://jericho.htmlparser.net/docs/index.html. Accessed 30 Nov 2016
37. Rigby PC, Robillard MP (2013) Discovering essential code elements in informal documentation. In: Proceedings of the 2013 International Conference on Software Engineering (ICSE' 13). IEEE Press, Piscataway, NJ, pp 832–841

38. Petrosyan G, Robillard MP, De Mori R (2015) Discovering information explaining API types using text classification. In: Proceedings of the 37th International Conference on Software Engineering – Volume 1 (ICSE' 15). IEEE Press, Piscataway, NJ, pp 869–879
39. Stack Exchange Inc (2015) Stack exchange data dump. https://archive.org/details/stackexchange. Accessed 30 Nov 2016
40. Oracle (2015) Java SE Development Kit 8 documentation. http://www.oracle.com/technetwork/java/javase/documentation/jdk8-doc-downloads-2133158.html. Accessed 30 Nov 2016
41. Sonatype Inc (2015) The central repository. http://search.maven.org/#search%7Cga%7C1%7Ccom.google.android. Accessed 30 Nov 2016
42. Dyer R, Nguyen HA, Rajan H, Nguyen TN (2013) Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: 35th International Conference on Software Engineering (ICSE' 13). IEEE Press, Piscataway, pp 422–431
43. (2016) The R Project for statistical computing. https://www.r-project.org/. Accessed 30 Nov 2016
44. Rocha AM, Maia MA (2016) Automated API documentation with tutorials generated from Stack Overflow. In: Proceedings of the 30th Brazilian Symposium on Software Engineering (SBES' 16). ACM, New York, pp 33–42
45. Lafetá RFQ, Maia MA, Röthlisberger D (2015) Framework instantiation using cookbooks constructed with static and dynamic analysis. In: Proceedings of the 23rd International Conference on Program Comprehension (ICPC'15). IEEE Press, Piscataway, NJ, pp 125–128
46. Henβ S, Monperrus M, Mezini M (2012) Semi-automatically extracting FAQs to improve accessibility of software development knowledge. In: Proceedings of the 34th International Conference on Software Engineering (ICSE'12). IEEE Press, Piscataway, NJ, pp 793–803
47. Moreno L, Aponte J, Sridhara G, Marcus A, Pollock L, Vijay-Shanker K (2013) Automatic generation of natural language summaries for Java classes. In: Proceedings of the 21st International Conference on Program Comprehension (ICPC'13). IEEE, Piscataway, NJ, pp 23–32
48. McBurney PW, McMillan C (2014) Automatic documentation generation via source code summarization of method context. In: Proceedings of the 22nd International Conference on Program Comprehension (ICPC'14). ACM, New York, pp 279–290
49. Sridhara G, Pollock L, Vijay-Shanker K (2011) Generating parameter comments and integrating with method summaries. In: Proceedings of the 19th International Conference on Program Comprehension (ICPC'11). IEEE Computer Society, Washington, DC, pp 71–80
50. Montandon JE, Borges H, Felix D, Valente MT (2013) Documenting APIs with examples: Lessons learned with the APIMiner Platform. In: Proceedings of the 20th Working Conference on Reverse Engineering (WCRE'13). IEEE, Piscataway, NJ, pp 401–408
51. Kim J, Lee S, Hwang S-W, Kim S (2009) Adding examples into Java documents. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'09). IEEE Computer Society, Washington, DC, pp 540–544
52. Monperrus M, Maia A (2014) Debugging with the crowd: a debug recommendation system based on Stackoverflow. Technical Report hal-00987395. INRIA. https://hal.archives-ouvertes.fr/hal-00987395/PDF/article.pdf
53. Treude C, Robillard MP (2016) Augmenting API documentation with insights from Stack Overflow. In: Proceedings of the 38th International Conference on Software Engineering (ICSE' 16). ACM, New York, pp 392–403
54. Dagenais B, Robillard MP (2012) Recovering traceability links between an API and its learning resources. In: Proceedings of the 34th ACM/IEEE International Conference on Software Engineering (ICSE' 12). IEEE Press, Piscataway, NJ, pp 47–57
55. van Deursen A, Kuipers T (1999) Building documentation generators. In: Proceedings of the IEEE International Conference on Software Maintenance (ICSM' 99). IEEE Computer Society, Washington, DC, p 40
56. Moonen L (2001) Generating robust parsers using island grammars. In: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE' 01). IEEE Computer Society, Washington, DC, p 13
57. Subramanian S, Inozemtseva L, Holmes R (2014) Live API Documentation. In: Proceedings of the 36th International Conference on Software Engineering (ICSE' 14). ACM, New York, pp 643–652