PeerJ Computer Science

Mining usage patterns for the Android API

Hudson S. Borges and Marco Tulio Valente

Department of Computer Science, Federal University of Minas Gerais, Belo Horizonte, Minas Gerais, Brazil

ABSTRACT

API methods are not used alone, but in groups and following patterns. However, despite being a key information for API users, most usage patterns are not described in official API documents. In this article, we report a study that evaluates the feasibility of automatically enriching API documents with information on usage patterns. For this purpose, we mine and analyze 1,952 usage patterns, from a set of 396 Android applications. As part of our findings, we report that the Android API has many undocumented and non-trivial usage patterns, which can be inferred using association rule mining algorithms. We also describe a field study where a version of the original Android documentation is instrumented with the extracted usage patterns. During 17 months, this documentation received 77,863 visits from professional Android developers.

Subjects Software Engineering
Keywords Application programming interfaces, Usage patterns, Android

INTRODUCTION

Methods in modern APIs are not used independently of each other, but according to some patterns (*Robillard et al., 2013*; *Long, Wang & Cai, 2009*). For example, the Android JavaDoc page that documents the `beginTransaction` method explicitly reports that it is usually used together with `setTransactionSucessful` and `endTransaction`. However, this page is an exception and most usage patterns are not documented at all. We reach this conclusion after inspecting the Android documentation, searching for 100 popular usage patterns, mined from a dataset of 396 applications. We found that only 12 patterns are somehow documented.

This paper reports the first (to the best of our knowledge) large-scale field study on the instrumentation of API documents with usage patterns. The study is based on the Android API, which is selected due to its complexity, size, and relevance to Android developers (*Syer et al., 2011*; *Ruiz et al., 2014*). We consider an API usage pattern as set of API methods that are used together with a certain frequency (*Robillard et al., 2013*). We extend a tool, called APIMiner (*Montandon et al., 2013*), to mine usage patterns from a dataset of Android open-source applications. This tool also instruments the original API documents with information on the extracted usage patterns.

The study reported in this paper is divided in three parts:

- First, we report a characterization study on the usage of the Android API by client applications. Our central goal is to check whether the Android API and the proposed dataset of Android clients are indeed interesting objects of study.

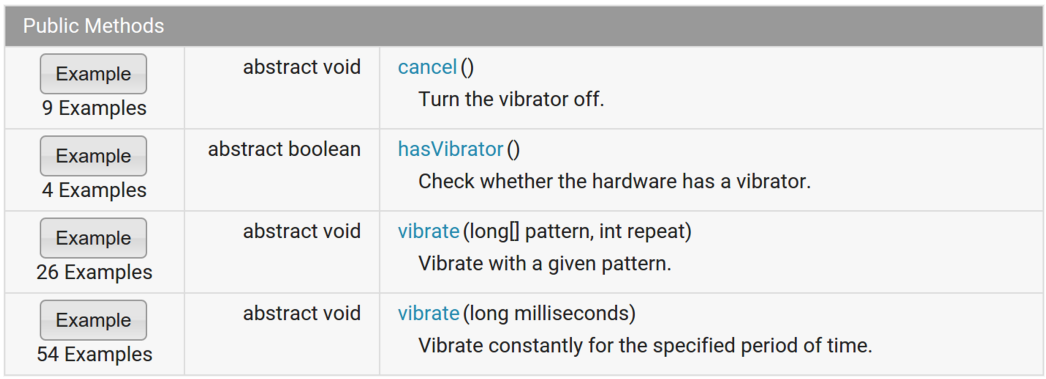
Submitted 14 May 2015
Accepted 9 July 2015
Published 29 July 2015

Corresponding author
Marco Tulio Valente,
mtov@dcc.ufmg.br

Academic editor
Philipp Leitner

Additional Information and Declarations can be found on page 13

DOI **10.7717/peerj-cs.12**



Public Methods		
Example 9 Examples	abstract void	cancel () Turn the vibrator off.
Example 4 Examples	abstract boolean	hasVibrator () Check whether the hardware has a vibrator.
Example 26 Examples	abstract void	vibrate (long[] pattern, int repeat) Vibrate with a given pattern.
Example 54 Examples	abstract void	vibrate (long milliseconds) Vibrate constantly for the specified period of time.

Figure 1 JavaDoc instrumented by APIMiner (*Montandon et al., 2013*).

- Next, we describe the methodology followed to extract usage patterns for the Android API and we characterize such patterns, in terms of their representativeness and complexity. We also show the results of a validation of the extracted usage patterns with two professional Android developers.
- Finally, we report a long-term field study, in which our version of the Android API instrumented with usage patterns and associated examples was made available to public access. During 17 months, it received 77,863 visits, coming from 160 countries. We analyze these visits to answer two research questions: (a) Do API users search for source code examples? (b) Do API users search for examples of usage patterns?

MATERIALS & METHODS

In this section, we present the tool we used to enhance API documents with information on usage patterns, the dataset used to mine these patterns, and the support and confidence thresholds used by the mining algorithm.

APIMiner

APIMiner (http://apiminer.org) is a tool that instruments JavaDocs with code examples, extracted from API clients (*Montandon et al., 2013*). As illustrated in Fig. 1, an "Example Button" is included in the original documentation, before the signature of each API method. By clicking on these buttons, developers are presented with source code examples for the documented API methods. A detailed presentation of the algorithms used by APIMiner to extract, summarize, and rank examples is out of the scope of this paper and we refer the interested reader to our previous work (*Montandon et al., 2013*).

For this article, we extend APIMiner with a capability to provide examples for API methods that are often called together. An API usage pattern has the following form:

$$M \Rightarrow M_1, M_2, \ldots, M_n$$

where M and M_i, $1 \leq i \leq n$, are methods from the API of interest. This pattern expresses that when the method M (antecedent term) is called by a given client method C, methods $M_1, M_2, \ldots, M_n$ (consequent terms) are usually called by C, not necessarily in this order.

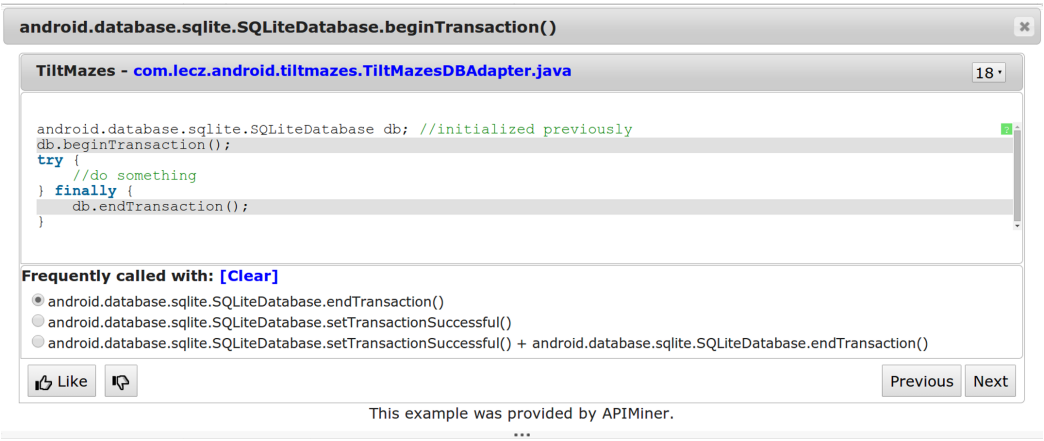


Figure 2 **Interface for presenting examples for API usage patterns.**

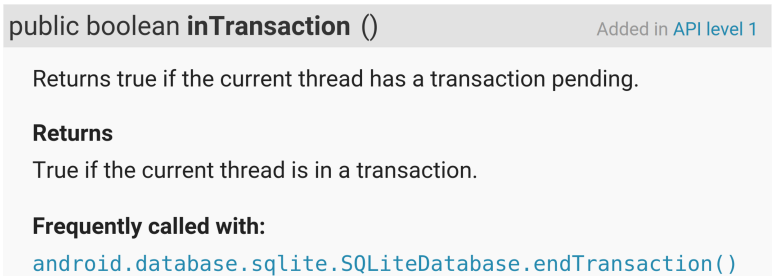
The extension adds a new module to APIMiner for inferring usage patterns. This module relies on the FP-Growth association rules mining algorithm, as implemented by the Weka data mining library (http://www.cs.waikato.ac.nz/ml/weka). The transactions used as input to FP-Growth are the methods of the client systems; the items are the Android methods called by these client methods (all calls are statically extracted, based on the static types of the target objects). Transactions with a single call to an API method are not relevant for our purposes. Therefore, they are discarded before executing the FP-Growth algorithm. To evaluate the significance of the extracted patterns we rely on two measures: support (number of transactions that include the methods in the pattern) and confidence (the probability of finding the methods from the consequent term in the subset of the transactions including the antecedent method).

We also extend the JavaDoc interface to show examples of usage patterns, as presented in Fig. 2. In the usage scenario illustrated by this figure, the API user initially requested examples for the `beginTransaction()` method, using the original interface provided by APIMiner. The window presenting the examples has a bottom panel with the usage patterns that have `beginTransaction` as the antecedent term. For instance, after selecting the first pattern the user sees examples that include not only `beginTransaction()` but also `endTransaction()`.

We also instrumented the JavaDoc sections that document the API methods with information on other methods they are frequently called with (if any), as presented in Fig. 3.

Mining dataset

Different from work that processes Android bytecode downloaded from Google Play (*Ruiz et al., 2014*), in our case it is important to have the original source code, to guarantee the extraction of examples with a minimum level of legibility. For this reason, we relied on GitHub to construct a dataset with the source code of Android systems, which we use for mining the usage patterns considered in this paper. We downloaded Android projects from GitHub that have at least 50 commits, to restrict the analysis to projects with


public boolean **inTransaction** () Added in API level 1

Returns true if the current thread has a transaction pending.

Returns

True if the current thread is in a transaction.

Frequently called with:

`android.database.sqlite.SQLiteDatabase.endTransaction()`

Figure 3 Usage patterns are presented in the detailed documentation of methods.

a minimum level of activity, and that are not forks of other projects, to avoid many similar projects in the dataset. By considering these requirements, we create a mining dataset with 396 projects, including well-known applications, such as WordPress, Astrid, K9, and ConnectBot. Considering all projects, the dataset includes 57,658 classes and 450,762 methods.

For this study, we use version 4.1.2-r1 of the Android API. We initially evaluate the usage of the API by the systems in the proposed dataset. Considering just the methods that call methods from the Android API, 59% call a second API method, which shows the feasibility of searching for usage patterns. Moreover, in the dataset, 40% of the public or protected methods from the Android API are never called.

We also analyzed the distribution of the number of API calls in our dataset. Figure 4 shows two histograms with the frequency of the number of calls. For each value n in the x-axis, the y-axis represents the number of methods with exactly n calls. To ease visualization, the first histogram shows methods with at most 40 calls; and the second histogram with at most 300 calls (in the full dataset, the number of calls ranges from 1 to 6,729). The histogram is right-skewed, meaning that while most methods are called few times (median equal to 5), we also have high and very high values. This finding implies that centrality and dispersion statistics measures (e.g., mean and standard deviation) should not be used to describe our empirical data on number of calls. Instead, the histogram suggests the data best fits a heavy-tailed distribution, possibly a power law. To check this possibility, we use the statistical framework proposed by *Clauset, Shalizi & Newman (2009)* for discerning power-law behavior in empirical data. By following this framework, we reject the (null) hypothesis that power-laws are a plausible explanation for our data, for a significance level of 5% (p-value $= 0.00$). However, this is not the same as concluding that the number of calls do not match a heavy-tailed distribution. In fact, Fig. 4 suggests a heavy-tailed behavior, possibly matching an alternative distribution (e.g., stretched exponential or log-normal).

Summary: Client methods tend to make multiple calls to the Android API (59% of the methods calling an API method, call at least a second one), which shows that there is space for mining methods that are frequently called together. However, the usage of the Android API follows a highly right-skewed distribution, which suggests that we should not expect to derive usage patterns for most API methods.

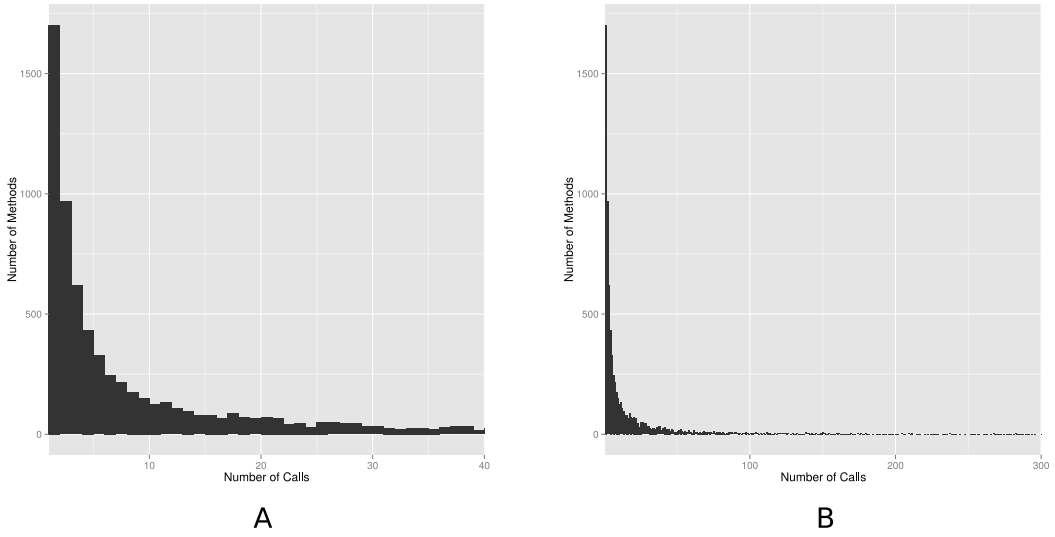


Figure 4 Histogram of number of calls. (A) Up to 40 method calls; (B) up to 300 method calls.

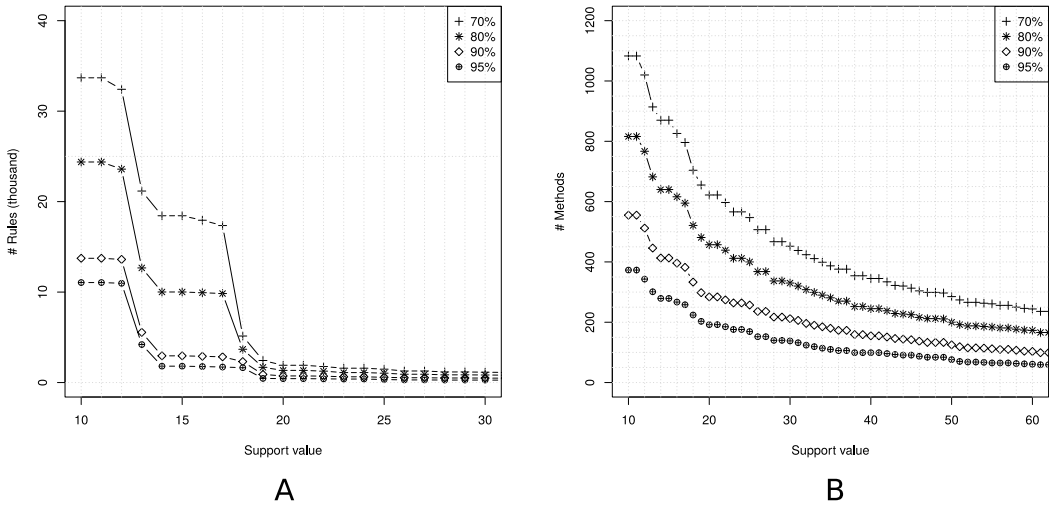


Figure 5 Setting up the support and confidence thresholds. (A) Number of rules vs support; (B) API methods coverage vs support.

Mining usage patterns

The first step for mining usage patterns is to set up the support and confidence thresholds. Figure 5A shows the number of association rules by varying the support values. We restrict the confidence thresholds to 70%, 80%, 90%, and 95% to keep a minimum quality in the rules. We can observe that small support values generate too many rules. Fixing a support of 10 transactions, we have 11,054 rules for a confidence of 95% and 33,685 rules for a confidence of 70%. In fact, the number of association rules starts to grow very rapidly for support values less or equal to 20. For this reason, we decide to use a support of 21 transactions.

Table 1 Usage patterns with highest support.

Usage pattern	Support	Confidence
`Activity.setContentView(View)`⇒ `Activity.onCreate(Bundle)`	1,362	75
`Toast.show()`⇒ `Toast.makeText(...)`	1,133	86
`ViewGroup.getChildCount()`⇒ `ViewGroup.getChildAt(int)`	1,077	75
`ViewGroup.getChildAt(int)`⇒ `ViewGroup.getChildCount()`	1,077	74
`AlertDialog.Builder.setTitle(...)`⇒ `AlertDialog.Builder.Builder(Context)`	1,019	98
`ContentValues.put(String,String)`⇒ `ContentValues.ContentValues()`	973	80
`AlertDialog.Builder.create()`⇒ `AlertDialog.Builder.Builder(Context)`	765	94
`AlertDialog.Builder.setMessage(...)`⇒ `AlertDialog.Builder.Builder(Context)`	736	96
`LayoutInflater.inflate(...)`⇒ `View.findViewById(int)`	697	72
`ContentValues.put(String,Integer)`⇒ `ContentValues.ContentValues()`	632	77

Figure 5B shows the number of methods covered by the extracted rules, considering just rules with a single method in the antecedent term. For a support of 21 transactions, the coverage ranges from 192 methods (confidence of 95%) to 624 methods (confidence of 70%). To increase API's coverage, we decide to fix a confidence of 70%.

By using the proposed thresholds, 1,952 usage patterns are mined, covering 624 API methods, which represent 5% of the public and protected methods in the Android API and 8% of the methods called in our dataset. To compute the patterns and the source code examples, it took around 10 h (on a Intel Xeon Six Core processor, with 64 GB RAM). However, it is important to highlight that this process is performed only once for a given API and corpus of client systems.

RESULTS

This section provides examples of usage patterns and also a classification of patterns in two categories, intra and inter-class. We also report an evaluation with developers, to validate whether the patterns really include methods that are called together. Finally, we report a field study, where we made our tool available to public usage.

Examples and types of patterns

Table 1 presents the usage patterns with the highest support. The most common usage pattern—found in 1,362 transactions with a confidence of 75%—models the computation required to create a focused UI window, which is called an `Activity` by the Android

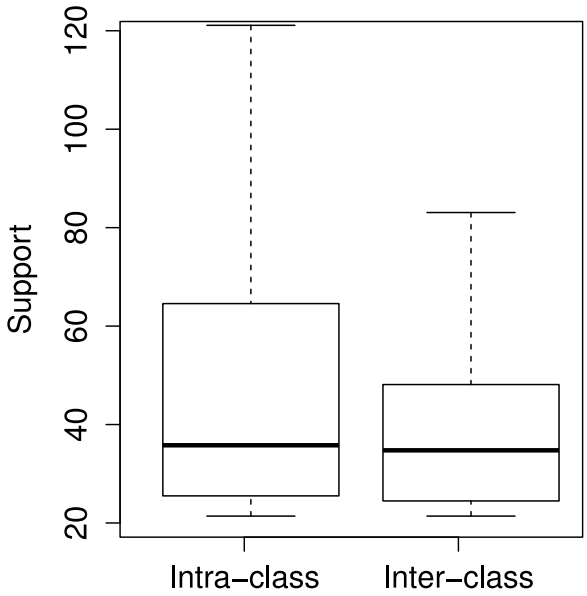


Figure 6 **Support values for intra-class and inter-class usage patterns.**

API. The pattern expresses that client methods that set an Activity's view by calling `Activity.setContentView(View)` usually call `Activity.onCreate(Bundle)` to initialize the view.

In Table 1, for nine patterns the methods in the antecedent and in the consequent term come from the same class. We refer to such patterns as intra-class usage patterns. It is easier to discover these patterns without tool support, since their documentation is restricted to a single JavaDoc page. However, considering the 1,952 usage patterns, there is an almost equal distribution between intra-class and inter-class usage patterns. Specifically, 50.3% of the usage patterns are inter-class, i.e., they have methods coming from more than one class. Figure 6 shows boxplots describing the distribution of the support values, regarding intra-class and inter-class usage patterns. The first and second quartiles of both distributions are very similar. Relevant differences start on the third quartile (63 methods for intra-class patterns vs 47 methods for inter-class patterns). For this reason, among the ten usage patterns in Table 1, only one is inter-class.

Evaluation by developers

Association rules can generate meaningless patterns, due to random relations that exist in most datasets. To validate the mined relations, we randomly selected a sample of 45 usage patterns, among the "less reliable" ones. By less reliable, we refer to patterns with the lowest support and confidence. This sample was selected among 477 usage patterns (≈25%) with support less than 50 and confidence less than 80%. We call this first sample the treatment group. Moreover, we randomly selected five sequences of methods, among the existing methods in the Android API. We call this second sample the control group. We presented each sequence of methods to two professional Android developers, with five and four years of experience in the Android API, respectively.

Table 2 Usage patterns evaluation by developers.

	Answers	
Developer	**Yes**	**No**
#1	33 (73%)	12 (27%)
#2	37 (82%)	8 (18%)
#1 && #2	28 (62%)	3 (7%)
#1 || #2	42 (93%)	17 (38%)

We asked the developers the following question:

Do you recommend to refer to method Y when documenting method X, since they are usually used together?

Both developers provided negative answers for all patterns in the control group, as expected. Table 2 summarizes the results for the 45 patterns in the treatment group. The first developer provided positive answers for 33 usage patterns (73%) and the second one for 37 patterns (82%). When combining the answers, 28 patterns (62%) were evaluated positively by both developers and 3 patterns (7%) received two negative answers. 42 patterns (93%) received at least a positive recommendation and 17 patterns (38%) received at least a negative recommendation. Therefore, according to the developers' judgment, we can reach a precision of 62% or 93% depending on the classification criteria (two positive answers vs at least one positive answer). This result reveals that the mined patterns are not coincidences or spurious relations, especially considering that we evaluated patterns in the first quartile of the support and confidence distributions.

However, it is important to highlight that the developer's judgment is influenced by their personal experience with the API. For example, we discussed in detail the patterns with two negative answers. In common, the developers initially argued they did not find reasons to the methods being called together. To clarify this common answer, we analyzed the following pattern in more details:

`Fragment.onActivityCreated(android.os.Bundle)` $\Rightarrow$ `Fragment.getActivity()`

One of the developers said this pattern does not make sense, "because the first method is a callback, called after an `Activity` is created." We then presented to this developer many instances of client code calling both methods. Most examples are subclasses of `Fragment` that override `onActivityCreated` and then call the superclass method, using `super`. After that, `getActivity()` is invoked to access the current `Activity` context. The developer acknowledged that some Android developers may use the methods in this way, but he also highlighted that the same "behavior can be achieved by arranging the `Fragment` classes in other ways," as he usually prefers to do.

Field study

We conducted a field study with the purpose of assessing the importance that API users give to usage patterns. Particularly, the study was not designed to evaluate the usability of

the provided source code examples, which it is always a challenge in the case of open field studies. Instead, we focused on the frequency that real users search for examples, including examples of usage patterns. We claim that answers for such questions—coming from the real usage of a complex API such as Android—are important for developers that want to instrument their API documents with usage patterns and to guide further research on code summarization techniques. In this way, the study aims to answer the following research questions: (RQ #1) Do API users search for source code examples? (RQ #2) Do API users search for examples of usage patterns?

To answer these two questions, we first made public an instance of APIMiner, at http://apiminer.org. We promoted this site in popular programming forums, like Hacker News and Reddit. However, we never controlled the access to the platform, i.e., the users had completely freedom to access any page of the API documentation, despite including or not usage patterns provided by API Miner. We collected access data to our site from May 13th, 2013 to October 14th, 2014 (17 months), using a private logging service and Google Analytics. During this time frame, APIMiner received a total of 77,863 visits, which gives an average of 150 visits/day. These visits came from 160 countries and the top three countries in number of visits were India (14.3%), United States (11.8%), and Brazil (4.9%). In total, 63,314 users visited the platform and 14,867 visits (19.1%) were from returning visitors. Finally, 54,704 visits (70.2%) came from queries performed in search engines, mostly in Google. The visits generated 114,124 page views. However, 60,029 page views (52.6%) have a very short duration, less than one second, or retrieved pages that are not instrumented by APIMiner (e.g., the site front page or several tutorials included in the Android documentation). Therefore, such page views were discarded from our analysis. Furthermore, when analyzing the requests for examples, we excluded requests to methods from the `SQLiteDatabase` class, because this class is used in the main page of the site to illustrate our usage patterns concept.

RQ #1: do API users search for source code examples?

The visits generated 14,402 requests for source code examples, i.e., clicks in the "Example Button." Therefore, on average 26.6% of the considered page views included an "Example Button" click. During the field study, 35,596 examples were presented to the users, i.e., on average 2.47 examples were presented after each click in the "Example Button." Therefore, the users navigated through the list of examples to see at least a second example. Table 3 presents the top ten methods with more requests for examples. These methods, with the exception of `Toast.setGravity(...)`, do not have usage patterns. However, among the 933 methods that received requests for example, 125 methods (13.4%) have at least one usage pattern.

RQ #2: do API users search for examples of usage patterns?

In our dataset, 624 methods have usage patterns. Considering these methods, 306 methods (49%) received at least one click in their respective "Example Button." Computing all clicks, these methods received 1,301 requests for examples. In other words, a window with source code examples including an option to present just examples for usage

Table 3 Top ten methods with the highest number of requests for examples.

Method	Req.
`ViewPager.OnPageChangeListener.onPageSelected(int)`	101
`SimpleCursorAdapter.SimpleCursorAdapter(Context,int,Cursor,...)`	91
`RectF.RectF(float,float,float,float)`	57
`Point.Point(int,int)`	56
`ViewPager.OnPageChangeListener.onPageScrollStateChanged(int)`	55
`SimpleCursorAdapter.setViewBinder(ViewBinder)`	50
`BitmapRegionDecoder.decodeRegion(Rect,BitmapFactory.Options)`	50
`RectF.RectF()`	44
`Toast.setGravity(int,int,int)`	43
`ViewPager.OnPageChangeListener.onPageScrolled(int,float,int)`	42

patterns—such as the one presented in Fig. 2—was activated 1,301 times during the field study. In such situations, the users selected an option to just show examples for a given usage pattern 399 times (30.6% of the window activations).

DISCUSSION

Our main findings on using APIMiner for extracting usage patterns are as follows.

Most Android API methods are underused

Only 60.5% of the methods in the Android API are called by the systems in our dataset. Therefore, even considering that this dataset might not represent an ideal sample of the whole population of Android applications (e.g., it only includes open-source apps), this result suggests that a considerable proportion of the Android API methods are rarely used by real clients. Therefore, API developers should monitor the usage of their API elements by real client systems. As a result, it is likely to conclude that many elements are underused or not used at all, suggesting a possible move to a streamlined API.

Less than 10% of the API methods called by clients have usage patterns

In our dataset, 8.1% of the Android methods have usage patterns. Even though this dataset does not include thousands of apps, as datasets based on Android bytecode (*Ruiz et al., 2014*), the considered support threshold was properly adapted to its size. Therefore, APIMiner shows that it is feasible to instrument API documents in a seamless way both with source code examples and with usage patterns. The only requirement is to have a representative sample of client systems. However, API developers should expect to retrieve usage patterns for less than 10% of their API methods.

In one out of three opportunities users search for usage patterns

During the field study, when examples for methods with usage patterns were available, in 30.6% of the cases the users requested examples for the mined patterns. Therefore, by

including examples of usage patterns, API developers can help such users on their specific needs when browsing API documents.

Threats to validity

There are at least four threats that could undermine the validity of our results. First, although the Android API is a complex and popular API, we cannot claim that our findings apply univocally to other APIs, especially to APIs for other languages or targeting a different application domain. Second, even in the universe of Android applications and considering a mining dataset with 396 projects, we might have missed usage patterns that are common just among applications from a particular category (e.g., location-based applications). Third, the selection of the support and confidence thresholds, as usual in association rules mining, is to some extent a subjective decision. To control this threat, we experimented various threshold combinations aiming to balance coverage and representativeness. Despite that, we could not estimate the impact on our field study of a different threshold selection, especially a less strict one. Fourth, our field study is an uncontrolled study, i.e., the subjects are not divided in treated and control groups. Therefore, there is always a risk of selection bias, especially when compared with controlled experiments. For example, we do not know the programming background of the developers that accessed our site (and therefore it is possible that all of them are experts on Android or the opposite). However, controlled experiments with professional developers are hard to conduct, especially in the area of API reuse. The main reason is that real-world development tasks may take hours or even days to be concluded (*Aparecido et al., 2011*).

RELATED WORK

Kagdi et al. compared frequent itemset and sequential-pattern matching and concluded that the latter is usually worth the additional cost (*Kagdi, Collard & Maletic, 2007*). However, they do not aim at the extraction of examples, when the order of the calls is immediately visible in the extracted code fragments. CodeWeb is a system that mines not only usage patterns regarding method calls, but also other forms of reuse, such as inheritance (*Michail, 2000*). PR-Miner (*Li & Zhou, 2005*) extracts general programming rules using frequent itemset mining, with focus on detecting buggy code. Code Recommenders (http://www.eclipse.org/recommenders) extends the Eclipse built-in Java API documentation with information such as the methods usually overridden when subclassing a selected type or the methods usually called on a selected object.

Systems such as Strathcona (*Holmes, Walker & Murphy, 2006*) and MAPO (*Zhong et al., 2009*) explore the syntactic context provided by the IDE to recommend examples. Strathcona extracts a set of structural facts about the context of a source code fragment. Then, it relies on this structural context to search in a pre-processed repository for code examples with similar structure. Finally, the best results are returned to the users for analysis. MAPO relies on a sequential pattern mining algorithm to provide source code examples for multiple API methods that are frequently used together in a pre-defined order. However, the examples do not have documentation purposes, because they are highly dependent on a particular development context. In contrast, systems such as APIMiner (*Montandon*

et al., 2013) and eXoaDocs (*Kim et al., 2013*) generate a new JavaDoc instrumented with source code examples. However, they do not provide support for API usage patterns. Altair is a tool that automatically generates API function cross-references, which are useful to populate *see also* sections in API documents (*Long, Wang & Cai, 2009*). The recommended functions are not computed using association rules, but based on their structural similarity with the function the cross-reference refers to.

Baker is a tool that links source code examples extracted from Q&A sites to API documentation. The tool relies on a constraint-based technique to uniquely identify fully qualified names in source code snippets (*Subramanian, Inozemtseva & Holmes, 2014*). ExPort is a tool that detects complex API usages, which can for example crosscut function implementations (*Moritz et al., 2013*). *Saied et al. (2015a)* propose a technique to detect multi-level usage patterns, which are API methods uniformly used across variable client programs, independently of usage context. The authors later proposed a technique to infer API usage patterns using structural and semantic relationships mined in the own API source code, i.e., without requiring client programs (*Saied et al., 2015b*). Since our central goal was to evaluate usage patterns in the field, with real API users, we decided to conduct our study using the most established usage patterns mining technique (association rules). Further work can include a comparison with the aforementioned techniques.

Studies on APIs usually report coverage rates similar to the one we found for API usage patterns. For example, *Ma, Amor & Tempero (2006)* investigated the usage of the Java API in a corpus of open-source software. They report that only about 50% of the classes and 21% of the methods in the Java API are used at all. A similar coverage appears on crowd-based Q&A forums, like Stack Overflow. For example, *Parnin et al. (2012)* evaluated the coverage of three APIs at the level of classes (and not of single methods) at Stack Overflow. They report that 13% of the classes in the Android API, 23% of the classes in the Java API, and 54% of the classes in the GWT API do not have discussion threads at all. Mojica et al. report a large-scale study showing the importance of the Android API. They report that 54% of the classes in Android apps inherit from a base class, compared to at most 36% in the case of non-mobile software (*Ruiz et al., 2014*). We also conducted a field study to evaluate the first implementation of APIMiner. We found a right-skewed distribution on the usage of the Android API, considering a dataset of 103 open-source Android systems (*Montandon et al., 2013*).

CONCLUSION

This paper provides a large-scale study of API usage patterns, including the extraction of patterns for a relevant API, an evaluation by expert developers, and a field study, when such patterns were presented to real users. For practitioners, especially API developers and maintainers, our study shows that with the wide availability of source code repositories, like GitHub, it is feasible to generate API documents instrumented with source code examples and usage patterns, both mined automatically. Moreover, the heavy-tailed behavior observed on the usage of API elements suggest to practitioners that most elements of their APIs may be underused or not used at all, and therefore it might be possible to

evolve to a streamlined API. Our study also shows that it is possible to collect real data on the usage of research prototypes. Further research is also possible, especially on techniques for summarizing code examples and for mining usage patterns. It would also be interesting to apply our technique to other APIs.

ADDITIONAL INFORMATION AND DECLARATIONS

Funding

This work was funded by grants from Brazilian National Research Council (CNPq) and Minas Gerais Research Foundation (FAPEMIG). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Grant Disclosures

The following grant information was disclosed by the authors:
Brazilian National Research Council (CNPq).
Minas Gerais Research Foundation (FAPEMIG).

Competing Interests

The authors declare there are no competing interests.

Author Contributions

- Hudson S. Borges conceived and designed the experiments, performed the experiments, analyzed the data, contributed reagents/materials/analysis tools, wrote the paper, prepared figures and/or tables, performed the computation work, reviewed drafts of the paper.
- Marco Tulio Valente conceived and designed the experiments, analyzed the data, contributed reagents/materials/analysis tools, wrote the paper, reviewed drafts of the paper.

Data Availability

The following information was supplied regarding the deposition of related data:

The dataset with the transactions and usage patterns used in this paper is available at: http://aserg.labsoft.dcc.ufmg.br/apiminer-dataset.

REFERENCES

Aparecido G, Nassau M, Mossri H, Marques-Neto H, Valente MT. 2011. On the benefits of planning and grouping software maintenance requests. In: *15th European conference on software maintenance and reengineering (CSMR)*. Piscataway: IEEE, 55–64.

Clauset A, Shalizi CR, Newman MEJ. 2009. Power-law distributions in empirical data. *Society for Industrial and Applied Mathematics Review* **51(4)**:661–703 DOI 10.1137/070710111.

Holmes R, Walker R, Murphy G. 2006. Approximate structural context matching: an approach to recommend relevant examples. *IEEE Transactions on Software Engineering* **32(12)**:952–970 DOI 10.1109/TSE.2006.117.

Kagdi HH, Collard ML, Maletic JI. 2007. Comparing approaches to mining source code for call-usage patterns. In: *4th workshop on mining software repositories (MSR)*. *Available at http://www.cs.kent.edu/~jmaletic/papers/MSR07-call-usage.pdf*.

Kim J, Lee S, Hwang S, Kim S. 2013. Enriching documents with examples: a corpus mining approach. *ACM Transactions on Information Systems* **31(1)**:1–27 DOI 10.1145/2414782.2414783.

Li Z, Zhou Y. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In: *13th symposium on foundations of software engineering (FSE)*. 306–315.

Long F, Wang X, Cai Y. 2009. API hyperlinking via structural overlap. In: *17th symposium on foundations of software engineering (FSE)*. New York: ACM, 203–212.

Ma H, Amor R, Tempero ED. 2006. Usage patterns of the Java Standard API. In: *13th Asia–Pacific software engineering conference (APSEC)*. Piscataway: IEEE, 342–352.

Michail A. 2000. Data mining library reuse patterns using generalized association rules. In: *22nd international conference on on software engineering (ICSE)*. 167–176.

Montandon JE, Borges H, Felix D, Valente MT. 2013. Documenting APIs with examples: lessons learned with the APIMiner platform. In: *20th working conference on reverse engineering (WCRE)*. Piscataway: IEEE, 401–408.

Moritz E, Linares-Vasquez M, Poshyvanyk D, Grechanik M, McMillan C, Gethers M. 2013. ExPort: detecting and visualizing api usages in large source code repositories. In: *28th international conference on automated software engineering (ASE)*. Piscataway: IEEE, 646–651.

Parnin C, Treude C, Grammel L, Storey M-A. 2012. Crowd documentation: exploring the coverage and the dynamics of API discussions on stack overflow. Technical report, Georgia Tech, College of Computing. *Available at http://www.cc.gatech.edu/~vector/papers/CrowdDoc-GIT-CS-12-05.pdf*.

Robillard MP, Bodden E, Kawrykow D, Mezini M, Ratchford T. 2013. Automated API property inference techniques. *IEEE Transactions on Software Engineering* **39(5)**:613–637 DOI 10.1109/TSE.2012.63.

Ruiz IJM, Adams B, Nagappan M, Dienst S, Berger T, Hassan AE. 2014. A large-scale empirical study on software reuse in mobile apps. *IEEE Software* **31(2)**:78–86 DOI 10.1109/MS.2013.142.

Saied M, Benomar O, Abdeen H, Sahraoui H. 2015a. Mining multi-level API usage patterns. In: *22nd international conference on software analysis, evolution and reengineering (SANER)*. Piscataway: IEEE, 23–32.

Saied MA, Abdeen H, Benomar O, Sahraoui H. 2015b. Could we infer API usage patterns only using the library source code? In: *23rd international conference on program comprehension (ICPC)*. *Available at http://www-etud.iro.umontreal.ca/~benomaro/publi/cwiaupulsc.pdf*.

Subramanian S, Inozemtseva L, Holmes R. 2014. Live API documentation. In: *36th international conference on software engineering (ICSE)*. New York: ACM, 643–652.

Syer MD, Adams B, Zou Y, Hassan AE. 2011. Exploring the development of micro-apps: a case study on the BlackBerry and Android platforms. In: *11th IEEE working conference on source code analysis and manipulation (SCAM)*. Piscataway: IEEE, 55–64.

Zhong H, Xie T, Zhang L, Pei J. 2009. MAPO: mining and recommending API usage patterns. In: *23rd European conference on object-oriented programming (ECOOP)*, vol. 5653. New York: ACM, 318–343.