Documenting APIs with Examples: Lessons Learned with the APIMiner Platform

João Eduardo Montandon, Hudson Borges, Daniel Felix, and Marco Tulio Valente

Department of Computer Science, UFMG, Brazil
{joao.montandon,hsborges,dfelix,mtov}@dcc.ufmg.br

***Abstract*—Software development increasingly relies on Application Programming Interfaces (APIs) to increase productivity. However, learning how to use new APIs in many cases is a non-trivial task given their ever-increasing complexity. To help developers during the API learning process, we describe in this paper a platform—called APIMiner—that instruments the standard Java-based API documentation format with concrete examples of usage. The examples are extracted from a private source code repository—composed by real systems—and summarized using a static slicing algorithm. We also describe a particular instantiation of our platform for the Android API. To evaluate the proposed solution, we performed a field study, when professional Android developers used the platform by four months.**

***Index Terms*—API documentation; source code examples; JavaDoc; field study.**

I. Introduction

Learning how to use most modern APIs is a challenging task given their ever-increasing size and complexity. As showed by vast empirical research, a major obstacle faced by developers when learning APIs is the lack of examples of usage [8], [11], [12]. In fact, the documentation of widely used APIs, such as those provided by the Java and Android platforms, basically consists of textual descriptions, with very few examples.

To help developers to use an API, we describe a platform—called APIMiner—that instruments the standard Java-based API documentation format with concrete source code examples of usage, extracted from a private repository. More specifically, this paper makes the following contributions:

- We report the design and implementation of APIMiner (Section II). Particularly, we describe a summarization algorithm—based on static slicing [17]—that extracts small but relevant source code examples (Section II-B).
- We describe a particular version of the APIMiner platform for the Android API, with 79,732 source code examples extracted from 103 open-source applications (Section III).
- We conducted a large-scale field study using the Android API version of the platform, when professional Android developers used the system by four months (Section IV). During this period, Android APIMiner was accessed 20,038 times from 130 different countries, generating more than 42,000 page views. Also, APIMiner provided 2,157 source code examples to its users.
- We document the main lessons learned after designing, implementing, and evaluating APIMiner (Section V).

II. APIMiner

This section describes the architetcure and the example summarization algorithm used by APIMiner.

A. Architecture

Many tools to recommend API source code examples are described in the literature (as summarized in Section VI). However, such tools—at least those compatible with JavaDoc-based documents—are usually research prototypes not mature enough to support an open field study. For this reason, we decided to invest in the implementation of our own tool for adding examples to JavaDoc-based documents. As illustrated in Figure 1, the architecture of our solution relies on three main components:

Source code repository: Developers usually view code examples as recommendations on how to use an API [11]. For this reason, APIMiner extracts examples from an internal and curated repository of source code projects, which must be populated before starting the instrumentation of the target API. In other words, to increase the quality of the provided examples, we decided to rely on an internal repository instead of automatically mining for examples in the web.

Pre-processing modules: To achieve scalability, APIMiner extracts, summarizes, and ranks the examples off-line, during a pre-processing phase. In this phase, we search the internal repository for methods that call the public methods provided by the target API. More specifically, two important tasks are performed. First, each method m that calls a given method m_{api} from the API is summarized, using a static slicing algorithm, as discussed in Section II-B. Second, the examples are ranked based on a weighted average of metrics that evaluate different perspectives of a software project. The first metric is the size of the example, in terms of lines of code (i.e., a source code metric). The premise is that small examples are better. The second and third metrics are respectively the number of commits of the file that provided the example in its original repository (i.e., a process metric) and the number of downloads of the target system (i.e., an usage metric). The premise in this case is that developers prefer examples that come from relevant files (which are changed many times) and from widely used and well-known systems. Finally, the examples are stored in a relational

978-1-4799-2931-3/13 © 2013 IEEE

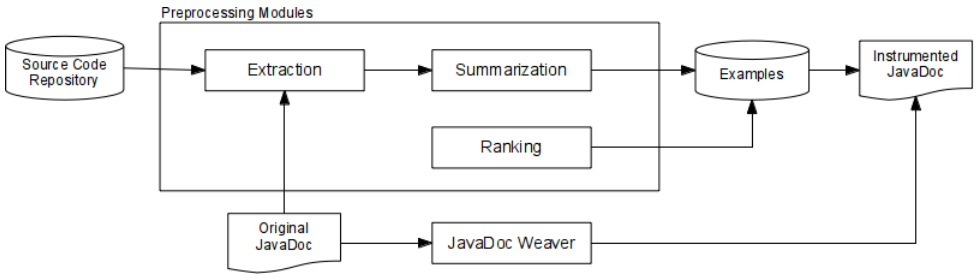

Figure 1. APIMiner architecture

database. In this way, a straightforward SQL query is used to retrieve the examples extracted for a given API method.

JavaDoc Weaver: In the proposed architecture, a documentation weaver tool automatically generates a new JavaDoc documentation, by inserting an "Example Button" in the original documentation, as illustrated in Figure 2.

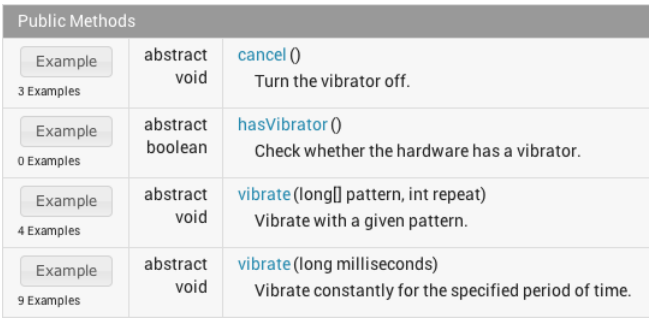

Figure 2. JavaDoc instrumented with "Examples Button"

B. Summarization Algorithm

Basically, source code examples must include their context, must have few lines of code, and must highlight the computation provided by the API [11]. Furthermore, a good example must be executed with minimal effort. To obtain examples with the aforementioned conditions, APIMiner relies on a summarization process that extracts the source code lines structurally related with a given API method call. For this purpose, we implemented an intra-method static slicing algorithm that performs the source code summarization.

Algorithm 1 presents this algorithm. Basically, the algorithm has three main functions: (a) `Summarization`; (b) `BackwardSlicing`; and (c) `ForwardSlicing`. It also uses the following auxiliary functions:

- `GetRdVars` and `GetWrVars`: return the variables that a given statement reads or writes to.
- `GetPrevStmts` and `GetNextStmts`: return the statements lexically located before or after a given statement.
- `GetParentStm`: retrieves the statement that is the lexical parent of a given statement, considering that the statements are represented by an Abstract Syntax Tree (AST).

The `Summarization` method represents the entry-point of the summarization algorithm. This method receives as input two arguments: (a) `seed`, which is the statement with the API method call; and (b) method `body`, representing all statements in the method where the seed was found. The method starts by declaring two local lists: `summarizedStmts`, which stores statements that have been processed and included in the example (line 2); and `selectedStmts`, which stores the statements that are relevant but that have not been processed by the algorithm (line 3). The algorithm then iterates over `selectedStmts` and executes the slicing process for each statement (lines 4-23). Finally, the `summarizedStmts` list is returned with the relevant statements (line 24).

In the main loop (lines 4-23), the algorithm gets a statement from `selectedStmts` and verifies whether it has been processed (lines 5 and 6). If it has not, the slicing algorithm is executed based on this statement, which is also stored in `stm` (lines 7-21). Basically, the algorithm first retrieves the variables `stm` reads and the statements located before it (lines 7 and 8). Then, we call the `BackwardSlicing` function with `rdVars` and `prevStmts` as input, and after that we store the result in `selectedStmts` for subsequent iterations (lines 9 and 10). If `stm` and `seed` are the same, the algorithm retrieves the variables that are written by `stm` and the statements located after `stm`. Then, we call the `ForwardSlicing` function with `wrVars` and `nextStmts` as input, and after that we store the result in `selectedStmts` for subsequent iterations (lines 12-15). Next, the algorithm verifies whether `stm` is nested in a control dependence block and, if true, retrieves this block and inserts it in `selectedStmts` for further processing (lines 17-20). Finally, the processed statements are stored in `summarizedStmts` (line 21).

Both `BackwardSlicing` and `ForwardSlicing` functions work in a similar way. They receive as input a list of variables, used to determine whether a statement is relevant and a list of statements to analyze. Then, both functions iterate over the statements list (lines 29-34 and 40-45), extract the variables of each statement (lines 30 and 41), and verify whether there is an intersection between the extracted variables and the variables received as parameter (lines 31 and 42). In case common variables are found, the statement is inserted in a list returned by the functions (lines 32 and 46). Although not presented for the sake of readability, the algorithm includes a last step

Algorithm 1 Summarization algorithm

```
1: function SUMMARIZATION(seed, body)
2:   summarizedStmts ← ∅
3:   selectedStmts ← seed
4:   while selectedStmts ≠ ∅ do
5:     stm ← Pop(selectedStmts)
6:     if stm ∉ summarizedStmts then
7:       rdVars ← GetRdVars(stm)
8:       prevStmts ← GetPrevStmts(stm, body)
9:       bStms ← BackwardSlicing(rdVars, prevStmts)
10:      selectedStmts ← selectedStmts ∪ bStms
11:      if stm = seed then
12:        wrVars ← GetWrVars(stm)
13:        nextStmts ← GetNextStmts(stm, body)
14:        fStms ← ForwardSlicing(wrVars, nextStmts)
15:        selectedStmts ← selectedStmts ∪ fStms
16:      end if
17:      if stm is a child of a control dependence then
18:        pStm ← GetParentStm(stm)
19:        selectedStmts ← selectedStmts ∪ pStm
20:      end if
21:      summarizedStmts ← summarizedStmts ∪ stm
22:    end if
23:  end while
24:  return summarizedStmts
25: end function
26:
27: function BACWARDSLICING(vars, statements)
28:  result ← ∅
29:  for stmt ∈ statements do
30:    stmtVars ← GetWrVars(stmt)
31:    if vars ∩ stmtVars ≠ 0 then
32:      result ← result ∪ stmt
33:    end if
34:  end for
35:  return result
36: end function
37:
38: function FORWARDSLICING(vars, statements)
39:  result ← ∅
40:  for stmt ∈ statements do
41:    stmtVars ← GetReadableVars(stmt)
42:    if vars ∩ stmtVars ≠ 0 then
43:      result ← result ∪ stmt
44:    end if
45:  end for
46:  return result
47: end function
```

where statements with an empty block (e.g., `for(...){}`) are removed from the returned slice.

Figure 3 shows in bold the result of a slicing regarding the call to the `vibrate` method (line 8). We can observe that the slice also includes the statement responsible for the target object declaration (lines 6-7)

```
1  public boolean onLongClick(View view) {
2      if (mIsSelecting) {
3          return false;
4      }
5      Log.i(AnkiDroidApp.TAG, "onLongClick");
6      Vibrator vibratorManager =
7       (Vibrator) getSystemSrv(Context.VIBRATOR);
8      vibratorManager.vibrate(50); // API method call
9      longClickHandler.postDelayed(...);
10     return true;
11 }
```

Figure 3. Summarization using static slicing (sliced code is in bold)

III. ANDROID APIMINER

Android applications are widely dependent on services provided by the Android API [14]. On average, 30% to 50% of the applications' code relies on the Android API. To ease the API learning effort, Google provides a detailed JavaDoc that documents the API. However, due to the lack of examples, the learning curve is still a problem for novice developers.

Therefore, Android provides an interesting API for evaluating APIMiner. For this reason, we implemented a particular instance of our solution for Android —called Android APIMiner. Android APIMiner provides 79,732 examples distributed in 2,494 methods (18% of the whole number of methods in the API), and 349 classes (19% of the whole number of classes in the API). The examples were extracted from 103 popular open-source systems, such as Wordpress and ZXing Barcode Scanner. As a prerequisite, we selected systems that attend three conditions: (a) they are implemented under open source licenses (such as GPL, Apache, etc); (b) they have a public source code repository; and (c) they must compile without errors (since the slicing algorithm works over the AST representation of the source code). Part of the systems was selected from a public Android open source application list.[1] The remaining systems were selected from curated developer websites (such as http://www.xda-developers.com, http://stackoverflow.com, etc) and specialized blogs (such as http://sudarmuthu.com). Table I shows a complete list with the systems included in our source code repository.

Table II shows the first 10 packages with more examples. As we can observe, the examples are highly concentrated, since the top 10 packages have 91% of the extracted examples (the remaining 9% are distributed in 74 packages). As expected, the packages in this table provide features commonly used by Android apps. For instance, `android.content` implements features related with content sharing and management and `android.widget`, `android.view`, and `android.graphics` are responsible for features related with GUI concerns.

Table III shows the top 10 classes in number of extracted examples. The listed classes have less examples than the packages in Table II: the top 10 classes have 54% of the extracted examples. Such classes also provide widely

[1] http://en.wikipedia.org/wiki/List_of_open_source_Android_applications

Table I
SYSTEMS IN THE SOURCE CODE REPOSITORY

System	System	System
4Chan Image Browser	Dialer2	robotfindskitten
aCal	Exchange OWA	Scrambled Net Full
ADW Launcher	FeedGoal	Secrets
Alien Blood Bath	FFVideoLive	Shortyz
Andless	Floating Image	Shuffle
Android Launcher	Formula	S. Tatham's puzzles
Android Metronome	Frozen Bubble	Sipdroid
Android motion	GCal Call Logger	SL4A
Android's Fortune	GCstar Scanner	Slashdot
AndroSens	GCstar Viewer	SMS Backup Plus
Andtweet	Hermit Android	Sokoban
Ankidroid	Hot Death	Solitaire Collection
Announcify	K9 Mail	Spell Dial
APG	Keepassdroid	Substrate
APNdroid	Lexic	SuperGenPass
Aptoide Client	LibreGeoSocial	Swallow Catcher
Aptoide Uploader	MandelBrot	Swiftp
ARViewer	MemorizingTrustMngr	Target
Audalyzer	MINDroid	Test Card
AR framework	Mnemododo	Tippy Tipper
Banshee Remote	Mustard	TouchTest
Barcode Scanner	Nethack Android	Tricorder
BatteryTracker	Newton's Cradle	Tumblife
Big Planet Tracks	OI About	Twisty
Broadcast Dumper	Open WordSearch	Twitli
Chime Timer	OpenMap framework	Vector Pinball
CIDR Calculator	OpenSudoku	Vidiom
Clusterer	Orbot	Voyager Connect
ConnectBot	Password Hash	Watch Aids
Contact Owner	Pedometer	Wiki Dici
CorporateAddressbook	Picture Map	Word Seek
Countdown Alarm	Plughole	Wordpress
Crowdroid	PMix	XBMC Remote
Cyanogen Updater	Replica Island	
Dazzle	Ringdroid	

Table II
TOP 10 PACKAGES IN NUMBER OF EXAMPLES

Package	# Examples
android.content	15,446
android.view	11,664
android.app	10,671
android.widget	9,493
android.os	7,016
android.util	5,710
android.graphics	4,216
android.database	3,648
android.preference	3,038
android.content.res	1,998
Total	72,900

used services, such as logging (`android.util.Log`), GUI (`android.view.View` and `android.widget.TextView`), and basic features (`android.app.Activity` and `android.content.Intent`).

A similar behavior is observed when we analyze the methods with more examples, as presented in Table IV. From an universe of 14,258 methods, the top 10 methods are responsible for 19% of the source code examples. Similarly, the listed methods implement services commonly used when developing Android applications, like methods for manipulating GUIs (`Activity.findViewById` and `setText`) and for implementing logging (`Log.d`, `Log.e`, `Log.i`, etc).

Table III
TOP 10 CLASSES IN NUMBER OF EXAMPLES

Class	# Examples
android.app.Activity	7,883
android.view.View	7,171
android.util.Log	5,599
android.content.Intent	3,840
android.content.Context	3,729
android.database.Cursor	3,612
android.widget.TextView	3,578
android.content.ContextWrapper	2,817
android.content.SharedPreferences	2,569
android.os.Bundle	2,490
Total	43,288

Table IV
TOP 10 METHODS IN NUMBER OF EXAMPLES

Method	# Examples
Activity.findViewById(int)	2,900
Context.getString(int)	2,024
TextView.setText(CharSequence)	1,908
Log.d(String,String)	1,454
View.setOnClickListener(View.OnClickListener)	1,326
View.setVisibility(int)	1,279
View.findViewById(int)	1,250
Log.i(String,String)	1,172
ContextWrapper.getResources()	1,001
Log.e(String,String)	979
Total	**15,293**

IV. FIELD STUDY

We conducted a field study using the publicly available version of the Android APIMiner platform. More specifically, we intended to answer the following questions:

1) How many users accessed Android APIMiner? How much time they spent in the platform? How many pages they visited?
2) Which locations do the visits to Android APIMiner come from?
3) How many examples Android APIMiner provided? What were the most requested examples?
4) Do developers search for source code examples?

To answer these questions, we analyzed usage access data collected from September 14th, 2011 to January 18th, 2012, in a total of four months. The data was obtained from two distinct sources: (a) Google Analytics service, which collected information related with APIMiner website access; and (b) a private logging service we implemented in the platform.

A. How Many Users Accessed Android APIMiner?

During the time frame considered in our field study, Android APIMiner received a total of 20,038 visits. As described in Table V, 14,412 visits (72%) originated from organic search—i.e., web search engine like Google, Bing, etc. Moreover, 3,393

visits (17%) originated from referral traffic, which means that the visitor was redirected to Android APIMiner from another web site (i.e., from links in blogs, forums, etc). The remaining 2,233 visits (11%) come from direct access to our platform.

Table V
TRAFFIC SOURCES

Traffic Origin	# Visits	% Visits
Organic search	14,412	72
Referral traffic	3,393	17
Direct access	2,233	11
Total	**20,038**	100

Figure 4 presents the number of visits to Android APIMiner by weeks. In general, we observe that the number of visits increased consistently. Also, there are two peaks of visits due to promotion posts on the Reddit social news site.[2] For example, in the second week of December, 2012, we posted a message about APIMiner in the Reddit's programming forum (`/r/programming`), which has around 415,000 readers.

Figure 4. Number of visits per week

During our field study, Android APIMiner received 42,034 pageviews, resulting on an average of 2.10 pages/visit. The users remained on the site for an average of 1:28 minutes.

B. Which Locations do the Visits to Android APIMiner Come From?

Table VI presents the top 10 countries in number of visits along with their Pages/Visit ratio. These countries were responsible for 12,029 visits, which represent 60% of the total. As we can observe, Android APIMiner was accessed from a large number of locations. However, three countries concentrate the access: United States (3,162 visits), India (2,086 visits), and Brazil (1,743 visits).

Table VI
TOP 10 COUNTRIES IN VISITS

Country	# Visits	Pages/Visit
United States	3,162	2.50
India	2,086	1.62
Brazil	1,743	3.28
France	855	2.65
Germany	827	2.26
Japan	777	1.71
United Kingdom	749	2.07
South Korea	719	1.63
Spain	577	1.75
Canada	534	2.26

[2]http://www.reddit.com

C. How Many Examples Android APIMiner Provided?

During the four months of our study, the users requested 3,910 examples for Android APIMiner. However, 1,753 requests (45%) were made for methods the platform has no examples. In other words, Android APIMiner has provided examples for 2,157 users requests (55%).

Figure 5 shows the 2,157 example requests handled by Android APIMiner distributed by weeks. The distribution is similar to the one presented in Figure 4. Furthermore, the peaks in this figure are also due to the posts at Reddit. The highest number of examples was provided in the second week of December, 2012 (due to a second post at Reddit). In this week, 722 examples were provided by Android APIMiner.

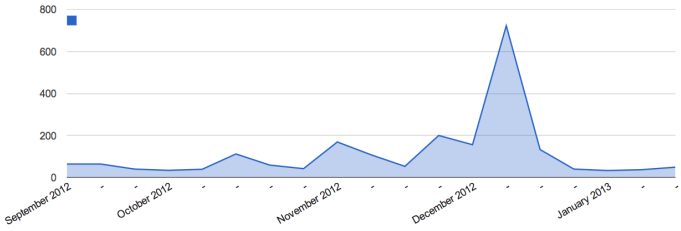

Figure 5. Number of examples provided per week

D. Do Developers Search for Source Code Examples?

To answer this question, we analyzed the queries used by the users when they reached Android APIMiner from a search engine. As mentioned, 14,412 visits originated from search engine queries. Due to privacy issues, Google does not provide search data from logged users (i.e., for search coming from users logged at Google services, like GMail). For this reason, we had to discard 9,774 visits. The remaining 4,638 visitors executed 3,660 different queries.

Table VII presents the top 10 most frequent user search queries. As we can observe, the top 10 queries were used 191 times (4%). Therefore, unlikely the results for the number of examples extracted per API method, the user queries present a diversified behavior.

Table VII
TOP 10 SEARCH QUERIES

Keyword	# Queries
speechrecognizer wait timeout	53
apiminer	30
datepicker.keep_screen_on	16
eglquerysurface egl_width android resize	15
listpopupwindow **example** android	15
android.net.rtp **example**	14
gridlayoutanimationcontroller **example**	13
android notificationcompat **example**	12
notificationcompat.builder **example**	12
fragmentactivity **example**	11
Total	191

We also counted the number of queries containing the `example` keyword. We found that 1,287 out of 3,660 available queries have this keyword (35%). When analyzing the top 10

queries, the `example` keyword is present in six queries. In summary, the `example` keyword was used frequently by the users, which reinforces our claim that developers search for source examples when accessing the documentation of an API.

V. Lessons Learned

We report here seven lessons we learned after designing, implementing, and evaluating APIMiner in the field.

Lesson Learned #1: APIMiner's architecture was essential to meet four fundamental requirements in this kind of tool: (a) support for production-quality code snippets (because the examples come from real systems, which were compiled before insertion in our source code repository); (b) seamless JavaDoc integration (because our weaver tool preserves the original interface and only inserts an "Example Button" in the original JavaDoc documentation); (c) scalability due to a pre-processing phase that extracts, summarizes, and stores the examples in a relational database; (d) support to on-the-fly updates to the examples database—for example, to include examples from new systems.

Lesson Learned #2: As illustrated by Figure 6, our results show that the usage of the Android API by client applications follows a power-law like distribution, which certainly makes it more difficult to provide a complete coverage of the API methods. In fact, similar degrees of coverage are observed in other forms of documentation. For example, crowd-based systems, such as Stack Overflow, also do not provide a complete coverage of the classes in the Android API [10].

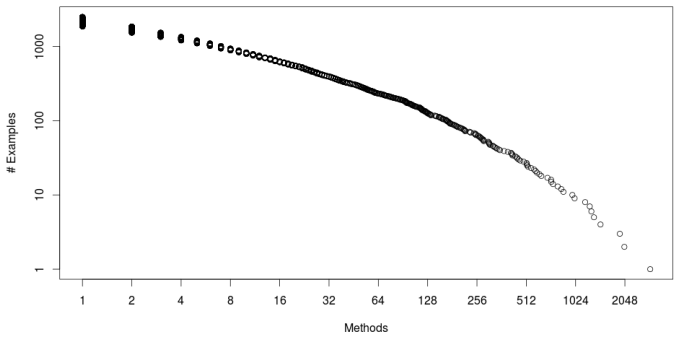

Figure 6. Number of examples for each API method (including only the 2,494 with at least a single example)

Lesson Learned #3: Even for a mature API as the Android API, it is not simple to retrieve a representative base of client systems. However, the aforementioned power-law like behavior implies that a small base of clients can provide examples to the most common API methods. Moreover, APIMiner's architecture makes it possible to increment this base without the need to regenerate the previously instrumented JavaDoc documents. Therefore, it is possible to continuously evolve and improve the database of examples.

Lesson Learned #4: Regarding their size, 75% of the extracted examples have less than ten lines of code, and 10% have between eleven and fifteen lines of code. Therefore, the slicing algorithm was effective in providing small examples.

Lesson Learned #5: Our usage data—including the number of examples provided by APIMiner and the number of search engine queries including the word example—reinforce the importance that developers give to examples when accessing API documents. On the other hand, it is time-consuming to provide examples for an extense API like Android, with almost 14K methods. For this reason, we claim that examples should be extracted automatically.

Lesson Learned #6: There is no correlation between the number of examples extracted for API methods and the number of examples requested by API users. Particularly, in Android APIMiner, the Spearman's rank correlation coefficient between extracted and requested examples is -0.04 (in a scale that ranges from -1 to +1). Therefore, when evaluating the examples coverage, it is important to consider not only the number of entries for a given method in the database of examples, but especially the number of examples the API users requested for this method.

Lesson Learned #7: Tools like APIMiner are useful not only to API users but also to API developers. More specifically, APIMiner helps API developers to understand how client code developers are using their APIs, by revealing the methods with more client calls or with more example requests. With this information in hand, API developers can improve the traditional documentation, in specific cases, with human-crafted examples. In other cases, this information may even motivate a redesign of the API (for example, to deprecate a method that is almost never used or to simplify the interface of methods with many example requests).

VI. Related Work

Many approaches have been proposed to provide source code examples automatically. Such approaches—referred in this section as API recommendation systems—can be organized in two distinct groups: IDE-based recommendation systems and JavaDoc-based recommendation systems.

IDE-based recommendation systems—such as Strathcona [4], MAPO [18], and API Explorer [3]—are implemented as IDEs' extensions (i.e., plug-ins). In general terms, the main advantage of these systems is their ability to explore the syntactic context provided by the IDE to recommend examples more relevant to developers (as in Strathcona). On the other hand, the examples provided by these systems typically cannot be used for documentation purposes, since they are highly dependent of a particular development context. Despite being an IDE-based recommendation system, MAPO relies on a sequential pattern mining algorithm to provide source code examples for multiple API methods thar are frequently used together in a pre-defined order. As our ongoing work, we are

Table VIII
COMPARISON BETWEEN APIMINER AND OTHER API RECOMMENDATION SYSTEMS

Category	System	Input	Interface	Output	Repository	Clustering	Ranking
IDE Tools	Strathcona	Statement	Plugin	Method	Private	—	✓
	API Explorer	Statement	Plugin	—	Private	—	—
	MAPO	Statement	Plugin	Method	Private	✓	✓
JavaDoc Tools	ExoaDocs	Method	Web	Slicing	Web	✓	✓
	APIExample	Type	Web/Plugin	Text	Web	✓	✓
	PropER-Doc	Type	Desktop	Method	Web	✓	✓
	APIMiner	**Method**	**Web**	**Slicing**	**Private**	—	✓

also extending APIMiner to include examples for multiple methods (as described in Section VII-C).

On the other hand, JavaDoc-based recommendation systems—such as APIExample [16], eXoaDocs [5], [6], and PropER Doc [7]—are implemented independently from any IDE and usually can be accessed from the web. As the key advantages, these systems have a wider reachability (because they are independent from other platforms) and greater scalability (because their results can be pre-processed). On the other hand, they usually do not provide the same level of precision as IDE-based recommendation systems.

Table VIII compares existing API recommendation tools with APIMiner. In fact, eXoaDocs is the tool closer to APIMiner. However, the process behind eXoaDocs' examples extraction—as well as the process used by the tool to instrument JavaDocs—is different from the one followed by APIMiner. For instance, eXoaDocs extracts examples from the web and summarizes them using only data dependencies. Moreover, the instrumented JavaDocs must be regenerated whenever a new example is processed. On the other hand, APIMiner relies on a curated source code repository and on a slicing algorithm that considers both data and control dependencies. Furthermore, APIMiner requires the insertion of a single button in a standard API documentation. Finally, we evaluated our platform in the field, using a complex and widely popular API.

VII. FINAL REMARKS

We conclude by presenting our contributions, limitations, ongoing work, and plans for future research.

A. Contributions

Our experience with APIMiner brings contributions both to API developers and researchers. For API developers, our experience is important because we documented the typical architecture and algorithms used in such systems. Moreover, we described the main challenges and benefits of making our tool publicly available for a popular API, like the Android API. For API researchers, APIMiner constitutes a real platform that provides a baseline for evaluating in the field new techniques for recommending API usage examples, particularly new summarization and ranking algorithms. Basically, in this case, we just need to replace (or extend) the current database of examples.

Android APIMiner is publicly available at:

http://aserg.labsoft.dcc.ufmg.br/apiminer

B. Limitations

APIMiner currently targets APIs documented in the JavaDoc format. Particularly, our slicing algorithm is tightly coupled to the Java language syntax and to the Eclipse AST. Moreover, our approach suffers from the "cold-start" problem, i.e., examples can only be extracted after the API is used by a reasonable sample of client systems.

C. Ongoing Work

Association rules: We have already implemented a new version of the APIMiner platform that provides examples for methods that are frequently called together. Basically, when an user requests an example for a method m, we also suggest the methods that are frequently called with m (in the same client code). The user can then select just examples including such methods. Figures 7 and 8 illustrate this new feature of the platform. Figure 7 shows an example for the `beginTransaction` method, but also indicates that this method is frequently called with other methods, like `setTransactionSuccessful`() and `endTransaction`(). Figure 7 shows an example for these three methods called together in the same client code.

We relied on an association rule algorithm to discover such patterns of simultaneous method calls [1]. The association rules were extracted from a set of 155 Android projects, available at GitHub. By mining this codebase, we extracted 1,952 patterns of frequent method calls (like the pattern presented in Figure 8). We also extracted 21,598 examples for such patterns. We are currently conducting a second field study with this new version of the platform, and for that reason we decided to center this paper in our first version.

IDE version: Many developers prefer to access JavaDoc documents directly from the IDE, without having to access a Web browser. For this reason, we are also working on a static version of API Miner for Eclipse and Android Studio. Basically, we changed our weaver tool to insert directly in the JavaDoc web pages examples of usage. Since these pages are rendered by the IDE, developers cannot navigate by the examples. Instead, the pages include a fixed number of examples, previously instrumented by the weaver tool.

D. Future Work

As future work, we intend to work in four fronts:

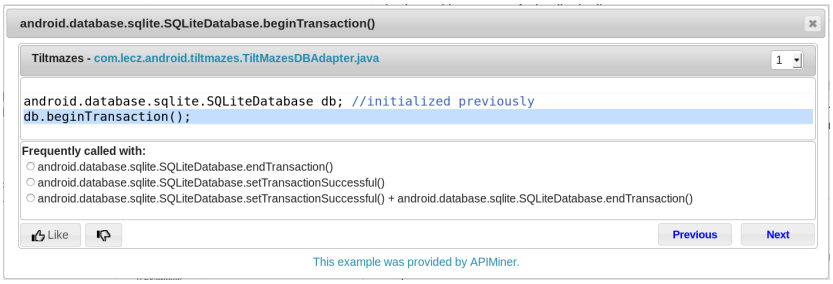

Figure 7. Example for `beginTransaction()`

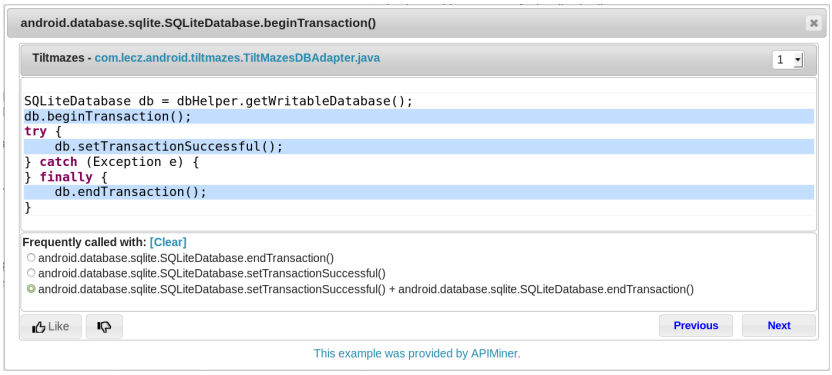

Figure 8. Example for `beginTransaction()` plus `setTransactionSuccessful()` and `endTransaction()`

- We intend to better evaluate the quality and the usefulness of the examples provided by APIMiner, possibly by means of controlled studies with developers.
- We intend to correlate and compare our results with studies that consider examples provided by crowd-based systems, like Stack Overflow [9], [10], [13].
- We plan to use APIMiner to provide examples to another API, possibly a subset of the Java API or the Eclipse API, using as the source code repository a compiled version of the Qualitas Corpus [15].
- We plan to investigate new summarization algorithms, like the approach proposed by Buse et al. [2].

Acknowledgments

Our research is supported by FAPEMIG, CAPES, and CNPq.

References

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *20th International Conference on Very Large Data Bases (VLDB)*, pages 487–499, 1994.

[2] Raymond P. L. Buse and Westley Weimer. Synthesizing API usage examples. In *34th International Conference on Software Engineering (ICSE)*, pages 782–792, 2012.

[3] Ekwa Duala-Ekoko and Martin P. Robillard. Using structure-based recommendations to facilitate discoverability in APIs. In *25th European Conference on Object-Oriented Programming (ECOOP)*, pages 79–104, 2011.

[4] Reid Holmes, Robert Walker, and Gail Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32(12):952–970, 2006.

[5] Jinhan Kim, Sanghoon Lee, Seung won Hwang, and Sunghun Kim. Adding examples into Java documents. In *24th International Conference on Automated Software Engineering (ASE)*, pages 540–544, 2009.

[6] Jinhan Kim, Sanghoon Lee, Seung won Hwang, and Sunghun Kim. Towards an intelligent code search engine. In *24th Conference on Artificial Intelligence (AAAI)*, 2010.

[7] Lee Wei Mar, Ye-Chi Wu, and Hewijin C. Jiau. Recommending proper API code examples for documentation purpose. In *18th Asia Pacific Software Engineering Conference (APSEC)*, pages 331–338, 2011.

[8] Samuel McLellan, Alvin Roesler, Joseph Tempest, and Clay Spinuzzi. Building more usable APIs. *IEEE Software*, 15(3):78–86, 1998.

[9] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What makes a good code example?: A study of programming Q&A in StackOverflow. In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 25–34, 2012.

[10] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. Crowd documentation: exploring the coverage and the dynamics of API discussions on Stack Overflow. Technical report, Georgia Tech, College of Computing, 2012.

[11] Martin P. Robillard. What makes APIs hard to learn? Answers from developers. *IEEE Software*, 26(6):27–34, 2009.

[12] Martin P. Robillard and Robert DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16:703–732, 2011.

[13] Dennis Schenk and Mircea Lungu. Geo-locating the knowledge transfer in StackOverflow. In *International Workshop on Social Software Engineering (SSE)*, pages 21–24, 2013.

[14] Mark D. Syer, Bram Adams, Ying Zou, and Ahmed E. Hassan. Exploring the development of micro-apps: A case study on the BlackBerry and Android platforms. *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 55–64, 2011.

[15] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha. Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *Software Engineering Notes*, 38(5):1–4, 2013.

[16] Lijie Wang, Lu Fang, Leye Wang, Ge Li, Bing Xie, and Fuqing Yang. APIExample: An effective web search based usage example recommendation system for Java APIs. In *26th International Conference on Automated Software Engineering (ASE)*, pages 592–595, 2011.

[17] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–257, 1984.

[18] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and recommending API usage patterns. In *23rd European Conference on Object-Oriented Programming (ECOOP)*, pages 318–343, 2009.