API Deprecation: A Retrospective Analysis and Detection Method for Code Examples on the Web

Jing Zhou and Robert J. Walker
Laboratory for Software Modification Research
Department of Computer Science
University of Calgary
Calgary, Canada
{jing.zhou, walker}@lsmr.org

ABSTRACT

Deprecation allows the developers of application programming interfaces (APIs) to signal to other developers that a given API item ought to be avoided. But little is known about deprecation practices beyond anecdotes. We examine how API deprecation has been used in 26 open source JAVA frameworks and libraries, finding that the classic deprecate–replace–remove cycle is often not followed, as many APIs were removed without prior deprecation, many deprecated APIs were subsequently un-deprecated, and removed APIs are even resurrected with surprising frequency. Furthermore, we identify several problems in the information commonly (not) provided to help API consumers transition their dependent code.

As a consequence of deprecation, coding examples on the web—an increasingly important source of information for developers—can easily become outdated. Code examples that demonstrate how to use deprecated APIs can be difficult to disregard and a waste of time for developers. We propose a lightweight, version-sensitive framework to detect deprecated API usages in source code examples on the web so developers can be informed of such usages before they invest time and energy into studying them. We reify the framework as a prototype tool (DEPRECATION WATCHER). Our evaluation on detecting deprecated ANDROID API usages in code examples on STACK OVERFLOW shows our tool obtains a precision of 100% and a recall of 86% in a random sample of 200 questions.

CCS Concepts

•Software and its engineering → Software libraries and repositories; Documentation;

Keywords

API deprecation, deprecation practices, web-based documentation, Deprecation Watcher.

1. INTRODUCTION

Software developers use existing frameworks and libraries to save development time and effort. Application programming interfaces (APIs) are the exposed interfaces of underlying software items that are intended to be reused by such developers. Ideally, frameworks and libraries keep their APIs unchanged and only add new APIs or change the underlying implementation of existing APIs. In reality, frameworks and libraries evolve over time for the same reasons as any software [21], and this evolution sometimes causes non-backwards compatible changes to their APIs. Any sudden elimination of an API would cause difficulties for the API's consumers, so *deprecation* of an API temporarily maintains backward compatibility while urging API consumers to transition to better alternatives. Unfortunately, we know little about deprecation in practice, the problems that it will cause for developers, and what could be done to improve the situation.

Information about deprecation [8, 10, 14, 18, 31, 38, 40] is generally anecdotal or based on assumptions that are either without supporting evidence or of dubious generality. There has been much work on general API evolution, focusing on migrating client code away from breaking API [6, 8, 11, 14, 18, 26, 27, 29, 39, 41], but this says nothing about deprecation practices. Some work has considered developer reactions to deprecation [13, 16, 22, 24, 34], notes issues only in passing with deprecation [8, 18], or fails to consider how deprecation evolves [20]. Work has been proposed to link web-based code examples to concrete APIs [9, 37], but this fails to consider the presence of multiple, simultaneous API versions, only some of which contain deprecated items.

When an API is deprecated, it is assumed that API consumers will be aware of it. According to a survey of over 3000 MSDN developers, developers indicate that they learn about new APIs primarily through web search instead of via official documentation. Web searches often take a developer to sites like STACK OVERFLOW or software development blogs, which have a high coverage of API for many frameworks [28]. Those sites often offer concrete code examples on how to use APIs, complementing official documentation. However, those examples are rarely updated, becoming a source of confusion and frustration when they refer to deleted or deprecated APIs. Even the official documentation is not always updated: for example, the ANDROID method `startManagingCursor(Cursor)` in class `android.app.ListActivity` has been deprecated since API level 11 yet it is still used in an example in the class documentation as of API level 21 [1].

Our work seeks to address these shortcomings: (1) by performing an in-depth, retrospective analysis to investigate deprecation practices in 26 open source JAVA systems (690 versions in total), and (2) by providing a tool, DEPRECATION WATCHER, that informs developers of the presence of deprecated APIs in web-based code examples. The tool aims to be fast and conservative, only announcing the presence of deprecated API usages when this is

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950298

definite. We evaluate our tool by comparing its ability to detect deprecated API usages in real posts on STACK OVERFLOW involving the ANDROID API, with classifications performed manually.

The remainder of the paper is organized as follows. Section 2 presents a motivational scenario for detecting web-based usages of deprecated APIs. Section 3 presents our retrospective study on API deprecation by tracking how API deprecation occurred and changed over time. In Section 4 we propose a lightweight, version-sensitive framework to identify deprecated APIs in web-based source code examples, implemented as our prototype tool DEPRECATION WATCHER. Section 5 presents our evaluation of DEPRECATION WATCHER. Section 6 presents a discussion about our work and its implications. Section 7 discusses related work

The contributions of this paper are: (1) a retrospective analysis of 26 open source JAVA systems over 690 versions in total to determine how deprecation is used in practice and over time; and (2) a lightweight framework to detect deprecated API usages in source code examples on the web, with its associated prototype tool.

2. MOTIVATION

Imagine an ANDROID developer who wants to capture a picture from a webview on an ANDROID device. Like a lot of developers [28], instead of searching in the official ANDROID documentation, he decides to look for help on the web. He switches from his work environment (ANDROID STUDIO) to a web browser and starts a GOOGLE search with the search terms "android capture picture webview." The top four results for this query returned by GOOGLE are all from STACK OVERFLOW.[1] The developer selects the first link [19] to see that the title of the question ("Capture picture from android webview") matches his query almost exactly and that the question was marked as solved. A quick glance at the webpage shows that the question has 4 upvotes, has been starred by 10 people, and that the accepted answer has 12 upvotes and was written by someone with a high reputation. The question has also been viewed more than 8200 times since it was asked. It seems like the accepted answer would address this developer's problem too (see Figure 1) so the developer decides to look into it.

After spending a few minutes to understand the code, the developer considers integrating it into his own work, as the code blocks seem self-contained, without external dependencies. Therefore the developer should be able to do little to no modification of his own code before he can start integrating the example code.

However, once he starts to integrate the code, ANDROID STUDIO warns him that the new code contains deprecated APIs. Specifically, three methods in the code example have been deprecated: setPictureListener() and capturePicture() in class android.webkit.WebView and onNewPicture() in class android.webkit.WebView.PictureListener. The setPictureListener() and onNewPicture() methods were deprecated in API level 12 and capturePicture() in API level 19, but the developer uses API level 21. The developer decides to find the replacements for the deprecated APIs, starting with setPictureListener(). This time, he resorts to the official ANDROID documentation to see if this API has been replaced by another one. However, the official documentation [2] for method setPictureListener() merely states, "This method was deprecated in API level 12. This method is now obsolete."

The developer again decides to search the web to see how others might have solved this problem. He uses the keywords "android setpicturelistener obsolete" to do another search. However, this time the top result from GOOGLE is a STACK OVERFLOW post [7] that merely refers back to the official documentation.

[1]All data collected as of 21 September 2015.

```
web = new WebView(this);

web.setPictureListener(new PictureListener(){

    public void onNewPicture(WebView view, Picture picture) {
      if(picture != null)
      {
       try
       {
        Bitmap bmp = pictureDrawable2Bitmap(new PictureDrawable(picture));
        FileOutputStream out = new FileOutputStream(filename);
        bmp.compress(Bitmap.CompressFormat.PNG, 90, out);
        out.close();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
      }
});

web.setWebViewClient(new WebViewClient() {
    public void onPageFinished(WebView webview, String url) {
        Picture picture = webview.capturePicture();

    }
});


web.getSettings().setJavaScriptEnabled(true);

setContentView(web);
```

12

helper function

```
private static Bitmap pictureDrawable2Bitmap(PictureDrawable pictureDrawable){
        Bitmap bitmap = Bitmap.createBitmap(pictureDrawable.getIntrinsicWidth()
        ,pictureDrawable.getIntrinsicHeight()
        , Config.ARGB_8888);
        Canvas canvas = new Canvas(bitmap);
        canvas.drawPicture(pictureDrawable.getPicture());
        return bitmap;
    }
```

Disclaimer: I haven't actually compiled this, it should work though

share improve this answer

edited Apr 9 '13 at 17:48 Garry 1,342 • 2 • 12 • 29

answered Oct 9 '11 at 10:54 Reno 24.9k • 6 • 56 • 82

Figure 1: Answer addressing developer's need? [33]

Having wasted much time, the developer decides to undo the changes to his code, and return to his first search instead of looking for replacements to the deprecated APIs. He clicks the second link in the GOOGLE search results [23]. The answer to the question states that the answer was found in another STACK OVERFLOW post. So again the developer goes to that answer. After evaluating the code example here, he decides to use it and he succeeds.

At this point, the developer has wasted a significant amount of time and energy, making multiple false starts. If he had been able to know whether a code example contains usages of deprecated APIs and furthermore, which APIs used there are deprecated, he could have either avoided spending time understanding the code example by immediately searching anew for alternatives or begun looking for the replacements of the deprecated APIs early on.

3. A RETROSPECTIVE ANALYSIS

Our first goal is to study how API deprecation has been used in JAVA third party frameworks and libraries. In particular, we focus our study on the deprecation of API methods and constructors because they tend to change more often than other API kinds (as seen in anecdotes and informal investigations). Henceforth, "deprecated APIs" refers exclusively to deprecated API methods and constructors unless otherwise specified. We address five research questions:

RQ1. Is API deprecation underused?
RQ2. How are deprecated APIs documented?
RQ3. How often does un-deprecation occur?
RQ4. How often are deprecated APIs removed later?
RQ5. When do deprecated APIs get removed?

Section 3.1 describes our criteria to choose the systems used in the study. Section 3.2 describes our process to gather API deprecation data. And lastly, Section 3.3 describes our data analysis address-

ing the research questions. Detailed data, results, and graphics are available elsewhere.[2]

3.1 Selection of Frameworks and Libraries

We used MVNREPOSITORY,[3] a search engine for MAVEN projects on MAVEN CENTRAL REPOSITORY,[4] to select libraries used in this study. We selected only projects that are JAVA third party frameworks or libraries, based on the following criteria. First, we chose open source frameworks and libraries because their source and binary code are relatively easier to obtain than closed source ones; source and binary code of a framework or library can help us construct an API change history and pinpoint when exactly an API gets deprecated and when a deprecated API gets removed. Second, we chose frameworks and libraries that are well known and widely used by JAVA developers; changes to the APIs in those frameworks and libraries affect a large number of developers. All the frameworks and libraries we chose were from the most popular projects on MVNREPOSITORY.[5] The ranking is based on how many other projects on MAVEN CENTRAL REPOSITORY depend on it. Third, we chose libraries that have existed for at least 6 years so that API deprecations would have time to change in some manner.

We selected 26 JAVA open source third party frameworks and libraries, for a total of 690 versions, for our study. The average number of releases for the systems in our study is 26.5 while the average time span for the systems is 10 years. With the exception of three systems (JDOM, LOGBACK and NEKOHTML), the first versions of the systems all start from 1.0 or later in our study.

3.2 Data Gathering

To track deprecated APIs, we considered the questions: (1) when was an API deprecated, and (2) what happened to the API later?

Our process of gathering deprecation data follows. We downloaded as many versions of the libraries as possible from their official website if an official archive was available there. When an official archive was not available, we downloaded the source and binary code of the libraries from the MAVEN CENTRAL REPOSITORY instead. We used source and binary code because they are the most reliable sources of this information.

After obtaining the library versions, we used two tools to process them: (1) QDOX[6] (a parsing tool for JAVA code that can extract various information about the code entities therein) to parse the annotations and JAVADOC of the entities in every version of the libraries to get a list of deprecated APIs in each version; and (2) CLIRR[7] (a tool to check JAVA libraries for binary and source compatibility between versions) to find important API changes.

We note also that, on MAVEN CENTRAL REPOSITORY, a single framework or library can correspond to multiple artifacts, e.g., COMMONS COLLECTIONS corresponds to two different artifacts. We calculated API changes across such boundaries.

3.3 Data Analysis

RQ1. Is API deprecation underused?

We found that all the systems in our study deprecated some APIs during the studied development spans, illustrating that API deprecation in JAVA third party frameworks and libraries is a common practice. However, we also found that five systems only deprecated APIs in one version (COMMONS-LOGGING, MAVEN-MODEL, MAVEN-PLUGIN-API, NEKOHTML, SLF4J); the number of deprecated APIs in these versions is also small.

We found that the limited use of API deprecation *cannot* be attributed to the fact that these systems have already stabilized before reaching the version that is the first version used in our study. For instance, NEKOHTML only deprecated one API even though its first version used in our study is 0.1. It has also removed 32 APIs in 57 versions over the course of 12 years, which is in sharp contrast with the number of its deprecations. In the majority of the systems, removed APIs outnumber deprecated APIs significantly and in many cases the two sets do not overlap. We found that API deprecation is indeed underused since many APIs were removed without prior deprecation.

RQ2. How are deprecated APIs documented?

Deprecation messages are important for developers to decide what to do with deprecated APIs. A deprecation message that links to the replacement of the deprecated API helps developers migrate to the new one. A deprecation message can also offer other relevant information about the deprecation such as the rationale for the deprecation and when the deprecated API is expected to be removed.

To analyze how library developers use deprecation messages, we extracted the deprecation messages associated with every deprecated API; not all deprecated APIs came with a message but we found that the majority of them did. When an API does not have an associated message, we consider its deprecation message to be empty. We found that 12 of the 26 systems changed some of their deprecation messages in a later version. In such cases, we use the revised deprecation messages. We manually examined the deprecation messages to answer the following questions.

How often do deprecation messages refer to a concrete replacement? We manually classified deprecation messages into four categories based on whether a concrete replacement is given: *without replacement*, *no mention of a replacement*, *a vague replacement*, and *a concrete replacement*. The difference between the first two categories is that the deprecation message in the former specifically states that there is no replacement for the deprecated API while the latter signifies the absence of any statement. *A vague replacement* offers only a general sense for a migration path.

We found that, on average, only 55.1% of all deprecation messages offered concrete replacements for the deprecated APIs. Only two systems (COMMONS IO and NEKOHTML) offered concrete replacements for all of their deprecated APIs, but NEKOHTML only deprecated one API in total. For 27.0% (7 out of 26) of the systems, concrete replacements were specified for less than 20% of their deprecated APIs. Six out of the 7 systems did not refer to a concrete replacement for any of their deprecated APIs.

How often do deprecation messages offer explanations? Knowing why a certain API has been deprecated helps developers make informed decisions. Deprecation messages were classified into two categories: *with explanation* and *without explanation*.

We found that on average, only 9.1% of the deprecation messages offered an explanation for the deprecated API. In fact, close to half (11 out of 26) of the systems offered no explanation for any of their deprecated APIs. The GSON library has the highest percentage overall of messages containing an explanation, but still at a mere 50%. Our results show that developers of these systems tend not to explain their decisions in deprecation messages; since IDEs like ECLIPSE can display deprecation messages for deprecated APIs, developers fail to utilize the most convenient place to communicate their decisions.

[2] http://lsmr.org/data/deprecation-watcher/
[3] http://mvnrepository.com/
[4] http://central.sonatype.org/
[5] http://mvnrepository.com/popular, accessed in March 2015
[6] http://qdox.codehaus.org/
[7] http://clirr.sourceforge.net/

How often do deprecation messages specify a time frame for deletion? Knowing when an API will be removed in the future affects the priority that developers assign to various changes needed to their projects. We classify API deprecation messages into three categories: *no removal plan*, *a vague removal plan*, and *a concrete removal plan*. *A vague removal plan* merely states that the deprecated API will be removed in the future, while *a concrete removal plan* offers the specific future version.

We found that on average, only 5.7% of the deprecation messages in the systems offered a concrete removal plan for the deprecated APIs. The majority (19 out of 26) of the systems did not specify any plan for any of their deprecated APIs. Only 27.0% (7 out of 26) of the systems had concrete plans for the removal of some of their deprecated APIs.

In summary, our results for RQ2 show that API producers do not make use of deprecation messages well enough to help API consumers migrate from deprecated APIs. Less than a quarter of the systems specified all replacements of the deprecated APIs. Furthermore, most systems do not offer explanations for their API deprecations, nor specify when deprecated APIs will be removed.

RQ3. How often does un-deprecation of APIs occur?
APIs are sometimes un-deprecated. We tracked deprecated APIs in each library over time to see if they stopped being marked as deprecated in a later version. The results show that almost half (12 in 26) of the systems have un-deprecated some of their APIs during the time span examined in our study. By manually examining these cases, we observed that in most cases the un-deprecated APIs still existed at the end of the period.

In 3 systems (JUNIT, LOG4J, and XERCESIMPL), some APIs were first removed, then brought back and immediately marked as deprecated. We call this phenomenon *remove–resurrect–deprecate* in comparison to the classic *deprecate–replace–remove* cycle. We hypothesize that this indicates that library developers have falsely assumed that these APIs were not used by API consumers and went forward to remove them. Un-deprecation and the phenomenon of *remove–resurrect–deprecate* indicate that API producers lack sufficient knowledge of how their systems are being used.

RQ4. How often are deprecated APIs removed later?
We tracked deprecated APIs in the systems over consecutive versions; if they stop appearing in a version, they are deemed as removed. We found that 42.3% of the deprecated APIs were later removed, in sharp contrast to the 0% reported elsewhere [31], due to Raemaekers et al. having overlooked the JAVADOC @deprecation tag and artificial boundaries subdividing single projects within the MAVEN CENTRAL REPOSITORY. Further investigation found that systems differ greatly in this aspect: some removed all their deprecated APIs while the majority of them removed none. We categorize the systems into the following three cases.

All the deprecated APIs were removed. Only 15.4% (4 out of 26) of the systems removed all their deprecated APIs (COMMONS-COLLECTIONS, COMMONS-CONFIGURATION, MAVEN-MODEL, NEKOHTML). Two systems only had one or two API deprecations in total (MAVEN-MODEL, NEKOHTML).

None of the deprecated APIs have ever been removed. A total of 53.8% (14 out of 26) of the systems never deleted any of their deprecated APIs during the time periods in our study.

Some of the deprecated APIs were removed. The remaining 30.8% (8 out of 26) of the systems removed some of their deprecations. Deprecated APIs that are not removed yet in this group may or may not be removed in later versions.

The fact that more than half of the systems never removed any of their deprecated APIs suggests that the developers of these systems are highly concerned with backward compatibility and fear to introduce breaking API changes by removing deprecated APIs. In the extreme, there is even evidence that some API producers intend to *never* remove deprecated APIs if possible [12].

RQ5. When are deprecated APIs removed?
We chose to examine when deprecated APIs were removed in systems that have removed more than two deprecated APIs, because it is hard to generalize from systems that removed less than two deprecated APIs. Seven out of the twelve systems fulfill this condition. We categorized these systems based on when they removed their deprecated APIs: *before version 1.0*, *during transitions to major releases*, and *other versions*. Different frameworks and libraries have different conventions about what is a major release. For example, LOG4J considers 1.2 to be a major release.

All the systems removed deprecated APIs during transitions to a major release. JDOM removed its deprecated APIs before version 1.0 and during its transition to 1.0. Five systems removed their deprecations exclusively during their transition to major releases. Since no versions before 1.0 of these systems were used in the study, it is unclear if they removed deprecated APIs in these versions. LOG4J is the only system that removed deprecated APIs in non-major releases. It removed 3 out of its 85 deprecated APIs in two non-major releases: 1.1.3 and 1.2.1. Overall, 74.4% of the removed deprecated APIs were removed during the transition to major releases and 99.0% of the removed deprecated APIs were removed either before version 1.0 or during transitions to major releases. We hypothesize that API developers are concerned with the backward compatibility of their systems. We note an inconsistency, however, since we also found that many APIs were removed without being deprecated first.

Overall, our study shows that deprecation is treated sporadically and inconsistently both within single systems and across systems. The lack of migration paths, explicit removal plans, and rationales all limit the API consumer's ability to act both proactively and reactively. Given these problems, we could at least do a better job of informing them that an API is deprecated.

4. TOOL AND FRAMEWORK

Our second goal is to provide a tool for developers to warn about when an example uses deprecated API. Figure 2 shows a code example involving deprecated APIs under which a warning message has been inserted by our tool, called DEPRECATION WATCHER.

The version-sensitive framework underlying this tool (see Figure 3) consists of three components: (1) an extractor/profiler for code elements, (2) a version-sensitive API matcher, and (3) a visual feedback component. In DEPRECATION WATCHER, the code element extractor and visual feedback component work on the client as an extension to the GOOGLE CHROME web browser, while the version-sensitive API matcher works on a server.

4.1 Extractor/Profiler

The framework needs access to the source code example. Most modern web browsers allow external programs to extend their functionalities. With the permission of the web browser user, they make the HTML and DOCUMENT OBJECT MODEL (DOM) of the current webpage available to extensions.

To extract code elements from the source code examples, we first extract the code snippets on a webpage that are contained in <pre> or <code> HTML tags, which is the norm on the web today. We

Figure 2: DEPRECATION WATCHER identified the usage of a deprecated API in the code example and appended a warning message below it with the details.

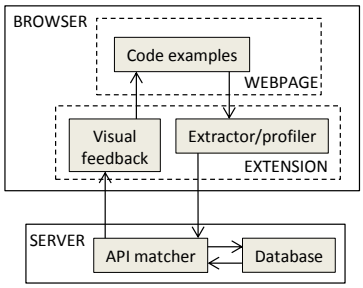

Figure 3: The structure of the framework.

```
1  protected Dialog onCreateDialog(int id) {
2      Dialog dialog;
3      switch(id) {
4          case 0:
5          dialog = new Dialog(this);
6          dialog.setContentView(R.layout.paused);
7          dialog.setTitle("Game Paused");
8          dialog.show();
9          break;
10     default:
11         dialog = null;
12     }
13     return null;
14 }
```

Figure 4: Example code containing a call to setContentView().

also process code snippets and reconstruct the lines in the code snippets before extracting code elements from them. The extraction is processed locally in the browser.

The next step is to construct a profile for every code element in the source code example (see Figure 4). To match code elements in the code snippets, we created JavaScript-based regular expressions (see Figure 5), matching them to every line in the code snippet

```
var typeVaribleRegex = /(\w+(?:\.\w+)?)\s(\b[a−z]\w{0,})/g;
var methodRegex = /(\b[a−z]\w+)\(/g;
var callerMethodRegex = /(\w+)\.(\w+)\(/g;
var argsRegex = /\((.∗?)\)/;
```

Figure 5: Regular expressions used to extract code elements. (For argsRegex, the number of arguments in any matched, non-empty string equals the number of commas plus one. Not used for nested calls.)

```
{ "kind": "method",
  "name": "setContentView",
  "cName": "Dialog",
  "args": 1,
  "type": "void",
  "tags": [ "android", "contacts" ] }
```

Figure 6: The profile for the example code element.

in order to construct a profile for them. The profile that is built for a code element depends on: (1) how much information can be extracted, given that many code examples on the web lack detailed information such as the fully qualified name of classes; and (2) the strategy that the API matcher uses to match a deprecated API since the API matcher uses the code element profile for matching. Many pieces of information about a code element could be used in the profile. For example, for setContentView() in the code example in Figure 4, our tool extracts the following types of information:

Kind of code element (kind). The kind of an API can be class, interface, method, constructor, or field. The kind of setContentView() is method because it matches the regular expression for a method call.

Name of code element (name). The name here is setContentView().

Name of enclosing class (cName). The name of the enclosing class for the code element is Dialog. This information is inferred through the context in the code. In line 2, a variable named dialog of type Dialog is declared, which is stored into the context for the code example. In line 6, setContentView(R.layout.paused) is invoked on dialog.

Argument count (args). The code element takes one argument.

Return type (type). The (inferred) return type of the code element is void.

Semantic tags (tags). The tags attached to the post are "android" and "contacts".

The profile for the code element setContentView() constructed with the information above is shown in Figure 6. The algorithm we use in the extractor/profiler is shown in Figure 7. (We have found that the tool works better when we also include code elements from the question posts as part of the context C_Q because answer posts sometimes refer to the types and variables used in the question.) The DEPRECATION WATCHER client-side asynchronously sends each profile constructed for an example to the server for matching.

4.2 Version-Sensitive API Matcher

The version-sensitive API matcher must attempt to map each profile to a concrete deprecated API in the target framework or library. The matcher in our tool matches against a repository of all APIs in the target system. The set of APIs in a framework or library can be obtained in several ways as mentioned in Section 3.2. Our API matcher is version-sensitive in that it returns its result based on the developer-configured version that the API matcher uses.

```
1  C_Q ← {}                                   ▷ context of question
2  C_A ← {}                                   ▷ context of answer
3  P ← {}                                     ▷ set of profiles
4  foreach s ∈ Q                              ▷ each snippet in question
5      do L ← RECONSTRUCT-LINES(s)
6         foreach l ∈ L
7             do C_Q ← C_Q ∪ EXTRACT-CONTEXT(l)
8  foreach A ∈ A                              ▷ each answer in answers
9      do foreach s ∈ A                       ▷ each snippet in answer
10            do L ← RECONSTRUCT-LINES(s)
11               foreach l ∈ L
12                   do C_A ← C_A ∪ EXTRACT-CONTEXT(l)
13                      (kind, name, cName, args, type, tags) ←
                                MATCHREGULAREXPRESSIONS(l)
14                      if type = NULL
15                        then type ← FINDTYPE(C_Q)
16                      p ← (kind, name, cName, args, type, tags)
17                      P ← P ∪ p
```

Figure 7: Algorithm to extract code elements and construct profiles for them.

On the server, we built a database of all deprecated API methods by parsing the DIFF files downloaded from the official ANDROID website. The DIFF files were produced by the ANDROID team using the JDIFF tool.[8] Each DIFF file contains the changes to the ANDROID API between two consecutive API levels. We stored information such as the name of the API, the enclosing type of the API, the deprecated version of the API, and the tag "android", into the database so an API could be matched with a code element profile easily. We also stored a database of all APIs in the newest version of the ANDROID API in the same way. Due to the fact that the code element profile only contains incomplete information of an API, we cannot be sure whether there is really a match when a code element profile matches a deprecated API in the database. Therefore we adopt a conservative strategy: we only consider there to be a match when *all* APIs matching the profile are deprecated.

The matcher can make use of a set of heuristics to match code element profiles to deprecated APIs in the database. Heuristics are necessary because it is difficult to obtain the fully qualified name of an API from a code example on the web directly, and sound analyses are likely to be expensive even when possible. We currently use the name of the code element, the simple name of the enclosing type, the number of arguments, and the return type as the elements in the profile that have to match exactly with a deprecated API to announce the usage of a deprecated API. The sets of tags have to have at least one element in common. After the matching process, the server returns all matches of deprecated APIs.

4.3 Providing Visual Feedback

The visual feedback component must parse the result and visually display the result to the user. The visual feedback could be designed and implemented in several ways [25]. First, deprecated APIs can be highlighted inline in the code snippet, mimicking IDEs where strikethroughs signify that that API has been deprecated. However, code snippets might already contain strikethroughs. Second, the feedback could be presented outside the code example. This option could offer more information about the deprecated APIs via text. Users then would have to look through the code to locate where the deprecated APIs are used. Finally, a hybrid of the two options is possible. For our tool, we chose the second option as least invasive.

When the extension receives the result for a code example from the server, it loops through all the APIs in the result and constructs an HTML segment containing the information about the deprecated APIs. The information includes the name of the API, the enclosing type of the API, the version in which the API is deprecated, and any deprecation message for the API. inserted.

In terms of running time performance, the non-optimized form of DEPRECATION WATCHER described here (as opposed to the batch processing form described below) completes its task within a maximum of five seconds on commodity hardware.

5. EVALUATION

For the evaluation of DEPRECATION WATCHER, we have two research questions:

> **RQ6.** How well does DEPRECATION WATCHER detect deprecated API usages in source code examples?
>
> **RQ7.** What factors affect how well DEPRECATION WATCHER performs?

5.1 Selection of Code Snippets

We chose to evaluate DEPRECATION WATCHER with posts containing code snippets on STACK OVERFLOW that are tagged "android". An accepted answer is deemed as the most helpful one by the asker on STACK OVERFLOW and is also the first answer a developer tends to look at when browsing the site. Many of these answers include source code examples to demonstrate how to accomplish certain tasks. In our evaluation, DEPRECATION WATCHER is configured to use ANDROID API level 21. If an API that was deprecated in or before level 21 is detected in a code example, DEPRECATION WATCHER will append a warning after the code example.

For batch processing: (1) we set up a database containing the documents (questions and answers) to be processed; (2) we fed the documents directly to the extractor/profiler; and (3) we replaced the visual feedback component with a simple collector that records whether deprecated API was detected or not for a given document. We chose our evaluation posts from STACK OVERFLOW's March 2015 data dump[9] with the following process. We first extracted all questions having an "android" tag and their answers from the data dump. Our database stores 638,756 documents in total. We filtered the documents using the set of all deprecated ANDROID APIs by the following two conditions. (1) The name of a deprecated API must appear in the code examples of the accepted answer. (2) The enclosing type name of that API must be mentioned in the document. These conditions leverage the lexical information in a document to filter out documents that *cannot* possibly be detected as having deprecated API usages by the tool, allowing us to focus our manual analyses on relevant cases.

We ended up with 7,464 documents that satisfied the filtering conditions. Some of the code snippets in the accepted answers of the documents may not contain deprecated API usages because our conditions for filtering are not strict enough. We then randomly selected 200 documents out of the 7,464 for our evaluation, as detailed, manual interpretation of each was needed at this stage.

5.2 Procedure

To evaluate DEPRECATION WATCHER, we reconstructed the URL for each question in the 200 documents from its question ID and we went to STACK OVERFLOW to examine the code snippets. We labelled them based on whether they contained deprecated ANDROID API methods, by manually checking the ANDROID official documentation. If the API was not present in the documentation, we went on to check the API DIFF files. (The absence of an API in the documentation was usually due to the API's deletion.)

[8] http://javadiff.sourceforge.net/

[9] https://archive.org/details/stackexchange

This manual process was quite time consuming. First, there can be two or more code examples in one answer. Second, in order to construct the signature of an API, a basic understanding of the code example is required. In many cases, it involves reading the question post and understanding the code snippets inside as well. Third, there is a significant amount of context switching involved between checking the ANDROID official documentation and examining the code snippets. As a result, the process of manually examining the code examples in a single answer can easily take 30 minutes.

5.3 Results

We used precision and recall to measure the performance of DEPRECATION WATCHER. If an API was identified by DEPRECATION WATCHER as deprecated and was manually classified as deprecated, it is classified as a true positive (TP). If an API was incorrectly identified as deprecated by DEPRECATION WATCHER, it is classified as a false positive (FP). If a deprecated API was not correctly identified as such, it is classified as a false negative (FN).

DEPRECATION WATCHER identified 74 deprecated ANDROID API usages in the source code examples, all of which were correctly identified. There were 86 deprecated ANDROID API usages in total in the code examples of the accepted answers in the 200 documents. It thus has a precision of 100% and a recall of 86% for those 200 documents. Considering the sample size, the population recall falls in the 95% confidence interval [81.19%, 90.81%].

We also found that, out of the 74 deprecated API usages that DEPRECATION WATCHER correctly detected, only 2 were acknowledged in the answers. Both of these were used deliberately *after* they were deprecated in order to target devices running older versions of ANDROID. Furthermore, only 3 out of all the 86 deprecated API usages were acknowledged by the answers or their comments. The third one, method removeGlobalOnLayoutListener (OnGlobalLayoutListener) in class android.view.ViewTreeObserver was used in a code example before it was deprecated and a comment pointed this out at a later date.

From documents that were filtered away, only negative results are possible; thus, the precision is representative of the entire document set assuming that the random sample was representative. Recall could be impacted by the documents that were filtered away if they contain false negatives. The first filtering condition retains all documents mentioning a name that matches deprecated API; therefore, only the second filtering condition could be an issue. If the enclosing type name is not mentioned in the document, there are two possibilities: (1) an unrelated type is being used and the matching name is a coincidence, but then the occurrence is a true negative; or (2) the enclosing type name is not mentioned. The second case could arise in an example whose enclosing type transitively or implicitly extends a type containing the deprecated API (direct and explicit inheritance would cause the document to be retained). Implicit inheritance would involve a sloppy example (contravening the STACK OVERFLOW policy on examples [36]) that is unlikely to result in useful comments and views, and so its practical impact would be low. The transitive inheritance could only involve custom classes, because the Android documentation lists all inherited entities within a class. But again, this seems unlikely since the example would be complex, contravening the STACK OVERFLOW policy.

5.3.1 What Factors Affect Performance?

We have found that almost all FNs in the evaluation arise from the fact that DEPRECATION WATCHER is limited in extracting information about the code elements in a code example. More specifically, some of the factors that affect how well the tool performs in recall are as follows.

```
int screenHeight = (short) Activity.getWindow().
      getWindowManager().getDefaultDisplay().getHeight();
int screenWidth = (short) Activity.getWindow().getWindowManager
      ().getDefaultDisplay().getWidth();
```

Figure 8: A code snippet with two chained method API calls.

```
int index = 0;
if (getLastNonConfigurationInstance() != null) {
    index = (Integer) getLastNonConfigurationInstance();
}
```

Figure 9: A code snippet with typecasting.

Chained method calls. This factor mostly affects the code element extractor. Since DEPRECATION WATCHER relies on simple regular expressions to extract API information, it is not able to extract the return types and the enclosing class of APIs inside a chained method call; an example of this is shown in Figure 8. DEPRECATION WATCHER failed to identify the deprecated methods getHeight() and getWidth() in this code snippet because it failed to identify the class they belong to. In the matching process, the API matcher tries to match the profiles of code elements getHeight() and getWidth(), only to find that several APIs with the same names exist in different classes and some of them are not deprecated. As a result, DEPRECATION WATCHER was not able to correctly map the code elements into deprecated APIs.

Class hierarchy. Our approach also ignores important type relationships like class inheritance in JAVA. This factor affects both the code element extractor and the API matcher. The code element extractor cannot identify the types some APIs belong to in some cases. For example, if a class A extends class B and calls a method m, DEPRECATION WATCHER would not know that method m comes from class B. The API matcher also matches types directly without taking the class hierarchy in the system into account. In JAVA an object can be type cast (explicitly or implicitly) to its subclass or superclass type. In Figure 9, the return type of API getLastNonConfigurationInstance() in class android.app.Activity has been typecast to Integer. As a result, DEPRECATION WATCHER misconstructs the code element profile for the code element here and consequently the API matcher was not able to return a match.

5.4 Evaluating Alternative Heuristics

The heuristics we used in the evaluation are the name, the number of arguments, the enclosing type, and the return type. Alternative heuristics could lead to a better result. But the example in Figure 9 shows how an extra heuristic may actually hurt the performance. In that specific example, the return type of the API getLastNonConfigurationInstance() was typecast to Integer. As a result, the tool misconstructs the profile for this code element and no match was found by the API matcher that tried to use all heuristics.

This prompted us to evaluate, after the fact, the performance on the same examples of three other combinations: (1) the name and the number of arguments; (2) the name, the number of arguments, and the return type; and (3) the name, the number of arguments, and the enclosing type. The results are shown in Table 1.

We found that the combination using only the name and the number of arguments performs as well as the combination that also includes the return type of a code element. Both of them were able to correctly identify a case that the other missed, but at the cost of missing a case that the other detected: the first set of heuristics was able to correctly detect code element getLastNonConfigurationInstance() in Figure 9 while the second set of heuris-

Table 1: Three more heuristic combinations were tried. N = name; A = # of arguments; R = return type; E = enclosing type.

Heuristics	TP	FP	FN	Precision	Recall
N, A	52	1	36	98.1%	59.1%
N, A, R	52	1	36	98.1%	59.1%
N, A, E	77	0	11	100.0%	87.5%

```
1 ProgramFragmentFixedFunction.Builder builder = new
      ProgramFragmentFixedFunction.Builder(mRS);
2 builder.setTexture(ProgramFragmentFixedFunction.Builder.
      EnvMode.REPLACE, ProgramFragmentFixedFunction.
      Builder.Format.RGBA, 0);
3 ProgramFragment pf = builder.create();
4 pf.bindSampler(Sampler.WRAP_NEAREST(mRS), 0);
```

Figure 10: A code snippet where return type aided detection.

```
public void updatePage(CurlPage page, int width, int height, int
      index) {
    switch (index) {
    case 0: {
        Bitmap front = loadBitmap(width, height, 0);
        page.setTexture(front, CurlPage.SIDE_FRONT);
        page.setColor(Color.rgb(180, 180, 180), CurlPage.
            SIDE_BACK);
        break;
    // ...
}
```

Figure 11: A case of incorrect detection.

tics was not. However, with the help of the return type information, the second set of heuristics was able to correctly detect a deprecated API in a code example [17] that the first set was not able to (see Figure 10): the return type ProgramFragment of the code element create() in line 3 helped to detect a usage of deprecated API.

The first two sets of heuristics also incorrectly detected a code element as deprecated API due to the fact that they do not make use of the enclosing type of the code element. (The relevant part of the code example [3] is shown in Figure 11.) The code element setTexture() was incorrectly detected as a deprecated ANDROID API even though it was from an external library because the type of code element page was not utilized.

We also found that the third combination of heuristics was able to outperform the heuristics we used to evaluate DEPRECATION WATCHER, despite it not making use of the return type.

We learned two lessons from evaluating these sets of heuristics. First, more heuristics do not necessarily lead to a better result. The addition of the return type of a code element as a heuristic does not help improve the performance of the tool when other heuristics are also applied. In the evaluation it proved to be useless when the enclosing type of a code element is included as one of the heuristics. This is somewhat expected since the return type is not considered as part of the method signature in JAVA and two APIs differing only in return type cannot co-exist in the same class.

Second, the detailed design of the heuristics matters. Had the code element extractor allowed for the return type to be typecast to another type, it would not have misconstructed a code profile in such cases. The return type would remain as a useless heuristic but would not have hurt the tool's performance. This has a greater implication: It can be hard to add complex heuristics even though in theory they may improve the performance, as complex heuristics tend to call for complex, error-prone implementations.

6. DISCUSSION

6.1 Why Lightweight Tooling Suffices Here

DEPRECATION WATCHER works well despite extracting only simple (essentially lexical) information from examples and performing only simple analyses thereon. We were initially surprised at how well this simple approach worked, expecting to encounter frequent situations where ambiguity arising from lack of semantic information would lead to poor recommendations. The fact that such problems occurred rarely can be attributed to the fact that code examples on STACK OVERFLOW are intentionally kept simple and self-contained so as to be easily understood. DEPRECATION WATCHER in its current form could not fare as well if general source code were analyzed. However, alternative analyses of greater complexity would slow down the tool, reducing its usability.

6.2 Deprecation Messages

We found that nearly half of the deprecation messages do not provide information on how to replace the deprecated APIs. A possible cause for this is that there is no standard format for deprecation messages, urging conformance from API producers. As a result, developers only include what *they think* is necessary. Therefore, it could be beneficial to establish a standard format and tool support for deprecation messages. Such standardization would also enable the creation of tools to analyze deprecation messages. For instance, to find all the deprecated APIs with a replacement and replace them automatically, or to generate a report for all the deprecations in a system and prioritize them based on information in the deprecation messages. We believe such standardization would not create much more overhead for developers of frameworks and libraries. API deprecation does not happen in every release of a system and when it does happen in a release, in most cases, not many APIs are deprecated simultaneously. Furthermore, we observed in the systems we studied, many APIs get deprecated due to the same, underlying change and can be removed in the future in the same version. A tool could use this information and reduce developers' workload.

6.3 Removing APIs

Removing APIs without prior deprecation seems dangerous because it can break client code without warning. Two factors may contribute to this phenomenon. First, it is likely unrealistic to deprecate APIs and retain them when a framework or library undergoes radical structural changes. Second, developers of frameworks and libraries remove APIs that they assume are not being used. When examining release notes, we noted that developers often make assumptions about how APIs in their systems are being used. (For example, in the release notes of COMMONS COLLECTIONS 4.0 [4], in the removed classes section, the developers state "removed unused class AbstractUntypedCollectionDecorator.")

Our results imply that developers are reluctant to remove deprecated APIs. However, when they do, the removal usually happens either before version 1.0 or during transition to major releases—and different systems have different conventions about what constitutes a major release. Many treat any release with an increase in the major release number as a major release while others treat releases with an increase in minor release number as major releases as well. Adopting semantic versioning [30] would be good for both API producers and consumers, providing greater clarity over the significance of new versions for planning by both parties.

6.4 Web Search in Software Development

Web search has proven valuable to developers during software development tasks. However, web search also has its problems: code

examples on the web can be written with different versions of an API, and developers can use different versions of the API as well. When APIs are deprecated or removed, some source code examples become outdated and are no longer helpful for developers who use the latest version of the API. At the same time, the same source code example can be of different value for developers who use different versions of an API. Web search engines take neither of these factors into consideration in their ranking of online resources.

We expect the usefulness of web search to deteriorate in the future if nothing is done to change this situation. Our work can be seen as an initial attempt to solve this problem. The framework we propose keeps track of what version of an API a developer uses in the API matcher and assesses whether a source code example on the web using the same API can be useful for the developer by detecting deprecated API usages.

6.5 Threats to Validity

The choice of libraries. We used a few criteria when selecting libraries to study. First, all the JAVA third party libraries we chose are open source projects. Therefore, our results do not necessarily reflect how API deprecation is used in closed source libraries. Second, all the libraries we chose are widely used by JAVA developers. As a result, our analysis may not be representative of less popular libraries. Third, all the libraries in our study have been in existence for at least 6 years. Therefore it may not reflect how new libraries are using API deprecation. Finally, we focused only on uses of the ANDROID API. Therefore this may not reflect uses of other APIs or web-based discussions thereof.

The versions of the libraries used in our analysis. We collected as many versions of the source code and binary code of the libraries as possible in our study so as to pinpoint when APIs were deprecated, removed, or un-deprecated. Unfortunately, we still were not able to locate some versions of some libraries. As a result, the exact version when the deprecation or deletion of certain APIs occurred might not be accurate. This is particularly relevant to RQ5 where we answered the question of when deprecated APIs were removed. However, by collecting as many versions of the libraries as possible, we were able to ensure any discrepancy between the apparent version and the actual version would be small at worst.

The use of STACK OVERFLOW. Our evaluation made use of only STACK OVERFLOW posts. Furthermore DEPRECATION WATCHER demonstrated strong performance because of the social limitations on examples discussed there. While the engineering aspects of the tool could easily be translated to similar Q&A sites, it is possible that the STACK OVERFLOW social limitations would not be in place. We believe that this is a dubious concern: more complex examples lead to type ambiguity would be just as problematic to humans on other sites, and thus there would be as much social pressure to "keep it simple, stupid."

The choice of posts in the evaluation. We selected posts that possibly contain deprecated APIs through the two criteria mentioned in Section 4. The first criterion filters out all posts that do not contain deprecated APIs. The second criterion aims to narrow the size of the posts down by another condition: the enclosing type name of an API should appear either in the question or in the accepted answer along with that API. This condition is important to filter out posts that contain APIs with a common name such as `create` because the majority of them are not deprecated. Another rationale behind this criterion is that knowing the type name an API belongs to is also important for a developer to understand the program. In rare cases, the enclosing type name of an API may be missing in both the question and answer text. In these cases, even if a document contains deprecated APIs, it would not end up in our evaluation set of posts.

6.6 Future Work

Investigate why APIs are removed without prior deprecation. We found evidence that API producers are concerned with backwards compatibility yet many APIs were removed without prior deprecation, possibly leading to broken client code. We think this paradox merits further investigation. We examined a few relevant release notes and found that API producers were aware of the fact that they were removing APIs without deprecating them first. However, they justified it with the belief that these APIs were not used and the removal of these APIs would not affect API consumers. In several systems we also found that most of the removed APIs in question were **protected** rather than **public**. API producers may believe that no user would inherit the classes in question and went forward to remove **protected** APIs. These hypotheses need further study.

Improve Deprecation Watcher. DEPRECATION WATCHER does well in precision because of its conservative API matching strategy. It only returns a match when it is certain. On the other hand, its performance in recall can be improved. We found that the main factor affecting its recall during our evaluation is that it does not deal with many parts of a language such as the class hierarchy. First, we can devise more complicated regular expressions to match structural information such as class inheritance and method overriding so more useful heuristics can be used; however, this alternative may complicate the implementation of the tool significantly. Alternatively, we could fully parse code examples to extract information; however, this alternative could reduce usability because of slowdown.

Evaluate with developers. We evaluated DEPRECATION WATCHER in terms of precision and recall, but with the involvement of developers, we could: collect direct evidence that deprecation problems waste time; consider the usability of the tool. For example, timely feedback to developers likely matters since this may affect perception of its usefulness. Furthermore, we could investigate what information is most useful to developers when a deprecated API is detected to improve the feedback component.

7. RELATED WORK

We consider two classes of related work more closely: deprecation practices and web-based code examples.

7.1 Deprecation Practices

Previous work from our lab has hinted at issues with API deprecation. Kapur et al. [18] found that deprecated entities do not always get deleted and deleted entities are not always deprecated; they manually studied only three systems and deprecation was not the focus. Cossette and Walker [8] found that replacement recommendations in deprecation messages were frequently missing or incorrect; again, their focus was not on API deprecation.

Hou and Yao [16] conducted a case study of the evolution of the AWT/SWING APIs to classify the intent behind API deprecations and additions, rather than how API deprecation is used.

Robbes et al. [34] studied the ripple effect of API deprecation in a SMALLTALK ecosystem and how developers react to it, finding that deprecation messages cannot be relied upon to be helpful because they can be missing or not offer concrete advice. Their study is done on only two SMALLTALK systems and might not be representative of how API developers use deprecation in general. They also focused on how API deprecation in one system affects other systems while our work focuses on how API deprecation is used and how deprecated APIs change over time within single systems.

Ko et al. [20] studied the quality of deprecation messages in eight JAVA libraries and found that 61% of the deprecation messages provide replacement APIs, while rationales and concrete examples on

how to use new APIs are rarely given. In contrast, our study also examines whether deprecation messages provide information on when deprecated APIs are expected to be removed since this information helps developers make decisions about when to migrate deprecated APIs. Furthermore, our study has a much larger scope.

Linares-Vásquez et al. [22] investigated how developers react to API changes by considering the volume of discussions about ANDROID API changes on STACK OVERFLOW: they found that removing API methods in particular triggers more discussions and from more experienced developers but API deprecation triggers discussion or even confusion as well.

Espinha et al. [13] examined how deprecation policy is used by several web service APIs and found that even when a web service gives a long deprecation timeframe before removing its old APIs, many developers still are not able to migrate their code in time. In the case of the GOOGLE MAPS API version 2, Google gave a deprecation timeframe of three years initially but had to extend it because many developers were not able to migrate their code in time. They reached the interesting conclusion that long deprecation periods leave developers too relaxed to migrate deprecated API code. However, their study was done on web service APIs while our work focuses on local APIs provided by libraries.

Raemaekers et al. [31] studied deprecation patterns in JAVA systems on MAVEN CENTRAL REPOSITORY. They found that deprecated APIs were never removed. However, they considered an API to be deprecated only if possessing a @Deprecated annotation, ignoring the presence of the @deprecated JAVADOC tag; furthermore, they failed to consider that projects there are sometimes subdivided artificially. Many researchers [10, 38, 40] seem to assume that deprecated APIs follow the *deprecate–replace–remove* cycle where the deprecated API is replaced by a new API and eventually gets removed, even if the cycle can take a long time. However, many deprecated APIs in various frameworks and libraries have not been removed despite having remained as deprecated for years; there is no indication that they will ever be removed. Furthermore, in JAVA and many third party libraries, some deprecated APIs have also been *un*-deprecated, which suggests the *deprecate–replace–remove* cycle is not the only possibility.

7.2 Web-Based Code Examples

Code examples are helpful for developers to learn APIs [35]. Commercial code search engines are able to search code from a vast number of open source projects that are available online. However, they are mostly keyword-based [32] which limits their utility. Several code search techniques [e.g., 5, 15, 32] have been proposed to improve code search by utilizing structural and semantic information of the code to make the results more relevant to developers. These techniques are only responsible for locating code examples and developers need to assess the code examples themselves. These code search techniques also value relevancy over timeliness.

Two approaches aim to link code elements in a code snippet to concrete APIs. RECODOC [9] links code-like terms in learning resources to concrete API elements, in order to solve the problem that the learning resources may not reflect API changes. The BAKER tool [37] tries to link the official documentation of a framework or library with example-based resources of the framework or library on the web so developers can benefit from both at the same time. Both approaches use partial program analysis and an oracle to link code elements to concrete APIs. Both approaches face the challenge of ambiguity when linking code elements in code examples to a concrete API in a framework or library; to uniquely identify an API, a fully qualified name is required while names in free text and code snippets tend to be ambiguous.

Although both approaches work well for their purposes, neither of them is able to address the problem of detecting deprecated API usages in source code examples on the web. BAKER assumes that API usages in code examples on the web are written with a specific version of a framework or library in mind and as a result can be mapped to that specific version of the API. This assumption is problematic for our context because code examples on the web can be written with different versions of the API of a framework or library in mind. This can be particularly true for new or rapidly evolving frameworks and libraries, e.g., the ANDROID API evolves at the rate of 115 API updates per month on average, many of which are deprecations or deletions [24]. Similarly, RECODOC only works on learning resources such as documentation and supporting channels that are known to correspond to a specific version of a framework or library. In contrast, our approach is designed to identify only deprecated APIs and does not assume that code examples on the web are written with a specific version of a framework or library; this permits DEPRECATION WATCHER to be lightweight.

8. CONCLUSION

Deprecation is an important tool for API producers and API consumers, but little is known about it beyond anecdotes and assumptions. We examined historical data of API deprecation and removal from 26 open source JAVA third party frameworks and libraries (690 versions in total). We found that, while deprecation is used by all the systems, it is underused: many APIs were removed without being deprecated first. Our analysis of deprecation messages shows their poor usability: just over half of the messages offered concrete replacements for their deprecated APIs; only 9.1% of the deprecation messages provided a rationale for the deprecated API; and a mere 5.7% of them specified a concrete timeframe for the removal of the deprecated API. We found un-deprecation in almost half of the systems we studied. We identified the phenomenon of *removal–resurrection–deprecation* of APIs within three systems as well. More than 40% of all deprecated APIs were removed but with uneven distribution: more than half of the systems never removed any of their deprecated APIs and only 15.4% of the systems removed all their deprecated APIs. Almost all of the deprecated-and-removed APIs were removed either before version 1.0 or during the transition to a major release; few APIs were removed elsewhere.

To address some of the difficulties that API consumers encounter in the face of deprecation, we proposed a version-sensitive framework to detect deprecated API usages in source code examples on the web. By utilizing the API change history of a target framework or library, we make our framework version-sensitive. We implemented the framework as a prototype tool called DEPRECATION WATCHER, which works as an extension to the GOOGLE CHROME web browser. It gives visual feedback when usages of deprecated API usages in a code example are detected. In our evaluation, DEPRECATION WATCHER achieved a precision of 1.0 and a recall of 0.86 in detecting deprecated ANDROID API usages in source code examples in the accepted answers on STACK OVERFLOW.

Our results have several implications. First, API producers need to provide better deprecation messages. Second, they need tool support to better understand how their APIs are used so they can have more confidence in removing deprecated APIs. Better support for API producers and API consumers would permit decreased risks for both groups.

Acknowledgments

This work was supported by a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada.

References

[1] Android. Documentation for class ListActivity. http://developer.android.com/reference/android/app/ListActivity.html, 2015. Accessed: 2015-07-20.

[2] Android. Documentation for class WebView. http://developer.android.com/reference/android/webkit/WebView.html, 2015. Accessed: 2015-09-21.

[3] AnilPatel. Top answer to "Android page curl with images streaming from web?". http://stackoverflow.com/a/16373967/1062364, 2013. Accessed: 2015-09-21.

[4] Apache Software Foundation. Apache Commons Collections: Release notes for v4.0. http://commons.apache.org/proper/commons-collections/release_4_0.html, 2015. Accessed: 2015-09-21.

[5] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming*, 79:241–259, 1 Jan. 2014. .

[6] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *Proc. IEEE Int. Conf. Software Maintenance*, pages 359–368, 1996.

[7] ciscogambo. "PictureListener is deprecated and obsolete, is there a replacement?". http://stackoverflow.com/questions/7166534/picturelistener-is-deprecated-and-obsolete-is-there-a-replacement, 2011. Accessed: 2015-09-21.

[8] B. E. Cossette and R. J. Walker. Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In *Proc. ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, pages 55:1–55:11, 2012.

[9] B. Dagenais and M. P. Robillard. Recovering traceability links between an API and its learning resources. In *Proc. ACM/IEEE Int. Conf. Software Engineering*, pages 47–57, 2012.

[10] D. Dig and R. Johnson. The role of refactorings in API evolution. In *Proc. IEEE Int. Conf. Software Maintenance*, pages 389–398, 2005.

[11] D. Dig, S. Negara, V. Mohindra, and R. Johnson. ReBA: Refactoring-aware binary adaptation of evolving libraries. In *Proc. ACM/IEEE Int. Conf. Software Engineering*, pages 441–450, 2008.

[12] dsaff. Response to "Request for clarification / roadmap / documentation #689". https://github.com/junit-team/junit/issues/689, 2013. Accessed: 2015-09-21.

[13] T. Espinha, A. Zaidman, and H.-G. Gross. Web API growing pains: Stories from client developers and their code. In *Proc. IEEE Conf. Softw. Maint. Reeng. Rev. Eng.*, pages 84–93, 2014.

[14] J. Henkel and A. Diwan. CatchUp!: Capturing and replaying refactorings to support API evolution. In *Proc. ACM/IEEE Int. Conf. Software Engineering*, pages 274–283, 2005.

[15] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Software Engineering*, 32(12): 952–970, Dec. 2006. .

[16] D. Hou and X. Yao. Exploring the intent behind API evolution: A case study. In *Proc. Working Conf. Reverse Engineering*, pages 131–140, 2011.

[17] A. Huerta. "What do the Renderscript FixedFunction shaders look like?". http://stackoverflow.com/a/9353027/1062364, 2012. Accessed: 2015-09-21.

[18] P. Kapur, B. Cossette, and R. J. Walker. Refactoring references for library migration. In *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 726–738, 2010.

[19] Karthi. "Capture picture from android webview". http://stackoverflow.com/questions/7702565/capture-picture-from-android-webview, 2011. Accessed: 2015-09-21.

[20] D. Ko, K. Ma, S. Park, S. Kim, D. Kim, and Y. Le Traon. API document quality for resolving deprecated APIs. In *Proc. Asia–Pacific Software Engineering Conf.*, volume 2, pages 27–30, 2014.

[21] B. P. Lientz and E. B. Swanson. *Software Maintenance Management*. Addison-Wesley, 1980.

[22] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk. How do API changes trigger Stack Overflow discussions?: A study on the Android SDK. In *Proc. IEEE Int. Conf. Program Comprehenension*, pages 83–94, 2014.

[23] lolyoshi. "How to capture a webview to bitmap in Android?". http://stackoverflow.com/questions/20900196/how-to-capture-a-webview-to-bitmap-in-android, 2014. Accessed: 2015-09-21.

[24] T. McDonnell, B. Ray, and M. Kim. An empirical study of API stability and adoption in the Android ecosystem. In *Proc. IEEE Int. Conf. Software Maintenance*, pages 70–79, 2013.

[25] E. Murphy-Hill and G. C. Murphy. Recommendation delivery: Getting the user interface just right. In M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, editors, *Recommendation Systems in Software Engineering*, chapter 9, pages 223–242. Springer, 2014.

[26] T. D. Nguyen, A. T. Nguyen, and T. N. Nguyen. Mapping API elements for code migration with vector representations. In *Proc. ACM/IEEE Int. Conf. Software Engineering*, 2016. To appear.

[27] M. Nita and D. Notkin. Using twinning to adapt programs to alternative APIs. In *Proc. ACM/IEEE Int. Conf. Software Engineering*, volume 1, pages 205–214, 2010.

[28] C. Parnin, C. Treude, and L. Grammel. Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow. Technical report, Georgia Institute of Technology, 2012.

[29] J. H. Perkins. Automatically generating refactorings to support API evolution. In *SIGSOFT Software Engineering Notes*, volume 31, pages 111–114, 2005.

[30] T. Preston-Werner. Semantic versioning 2.0.0. http://semver.org/. Accessed: 2015-08-10.

[31] S. Raemaekers, A. Van Deursen, and J. Visser. Semantic versioning versus breaking changes: A study of the Maven repository. In *Proc. IEEE Int. Wkshp. Source Code Analysis and Manipulation*, pages 215–224, 2014.

[32] S. P. Reiss. Semantics-based code search. In *Proc. ACM/IEEE Int. Conf. Software Engineering*, pages 243–253, 2009.

[33] Reno. Top answer to "Capture picture from android webview". http://stackoverflow.com/a/7703007, 2013. Accessed: 2015-09-21.

[34] R. Robbes, M. Lungu, and D. Röthlisberger. How do developers react to API deprecation?: The case of a Smalltalk ecosystem. In *Proc. ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, pages 56:1–56:11, 2012.

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

[36] Stack Exchange. How to create a minimal, complete, and verifiable example (revision 2016.3.10.3331). http://stackoverflow.com/help/mcve, 2016. Accessed: 2016-03-10.

[37] S. Subramanian, L. Inozemtseva, and R. Holmes. Live API documentation. In *Proc. ACM/IEEE Int. Conf. Software Engineering*, pages 643–652, 2014.

[38] K. Taneja, D. Dig, and T. Xie. Automated detection of API refactorings in libraries. In *Proc. IEEE/ACM Int. Conf. Automated Software Engineering*, pages 377–380, 2007.

[39] R. Štrobl and Z. Troníček. Migration from deprecated API in Java. In *Companion Conf. Syst. Progr. Lang. Appl. Softw. Humanity*, pages 85–86, 2013.

[40] T. Xie, M. Acharya, S. Thummalapenta, and K. Taneja. Improving software reliability and productivity via mining program source code. In *Proc. IEEE Int. Symp. Parallel Distr. Proc.*, pages 1–5, 2008.

[41] Z. Xing and E. Stroulia. API-evolution support with Diff-CatchUp. *IEEE Trans. Software Engineering*, 33(12):818–836, 2007.