API Deprecation: A Retrospective Analysis and Method for Detecting Deprecated API Usages in Code Examples on the Web

by

Jing Zhou

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA
January, 2016

© Jing Zhou 2016
Abstract

Developers increasingly resort to online resources in software development. Online resources often offer code examples that demonstrate which application programming interfaces (APIs) to use and how to use them in order to accomplish a task. However, APIs can become deprecated when a framework or library evolves, to indicate that they are obsolete, dangerous, and/or to be eliminated in future versions. Code examples on the web do not necessarily reflect such changes. Spending time to understand code examples that demonstrate how to use deprecated APIs can be a waste of time for developers. Meanwhile, API deprecation has not been systematically studied; existing work on API deprecation is incomplete or problematic.

We empirically study how API deprecation has been used in 26 open source Java frameworks and libraries. We find that not nearly enough information is provided to help users via deprecation messages. In sharp contrast with previous work, our results show that more than 40% of all deprecated APIs were removed, while more than half of the libraries never removed any. The removal of deprecated APIs almost exclusively occurred before version 1.0 or during the transition to a major release, suggesting developers take backward compatibility of their system seriously. On the other hand, API deprecation also seems to be underused, since many APIs were removed without being deprecated first.

We propose a 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 examining the code examples. We implemented the framework as a prototype tool named Deprecation Watcher. Our evaluation on detecting deprecated Android API usages in source code examples on Stack Overflow shows our tool works well, obtaining a precision of 1.0 and a recall of 0.86.
Acknowledgements

I would like to thank my supervisor Robert J. Walker for his guidance over the past two years, in which he taught me to think critically about research work both of my own and by others. He also generously provided financial support for my research, for which I am very grateful.

I would also like to thank Bradley E. Cossette for his help with my research.

Lastly, I would like to thank Hao Men for useful discussions about my research.
Dedication

I dedicate this thesis to my parents and Patrick. They have always supported my decisions.
# Table of Contents

Abstract ................................................................. i  
Acknowledgements .................................................. ii  
Dedication ............................................................... iii  
Table of Contents ................................................... iv  
List of Tables ......................................................... vi  
List of Figures ........................................................ vii  
1 Introduction ......................................................... 1  
  1.1 Background and Motivation ................................... 2  
  1.2 Research Questions and Objectives ......................... 8  
  1.3 Contributions .................................................. 9  
  1.4 Thesis Structure ............................................... 9  
2 Motivation .......................................................... 11  
  2.1 Motivational Scenario ....................................... 11  
  2.2 Summary ........................................................ 15  
3 Related Work ....................................................... 16  
  3.1 API Deprecation ............................................... 17  
  3.2 Source Code Examples on the Web .......................... 19  
  3.3 Summary ........................................................ 21  
4 Data Collection and Analysis ..................................... 23  
  4.1 Selection of Frameworks and Libraries .................... 23  
  4.2 Gathering Data ................................................ 24  
  4.3 Data Analysis .................................................. 26  
  4.4 Summary ........................................................ 38  
5 Model and Tool ..................................................... 40  
  5.1 Extracting Code Elements on the Web ..................... 42  
  5.2 Version Sensitive API Matcher .............................. 45  
  5.3 Providing Visual Feedback ................................... 48  
  5.4 DEPRECIATION WATCHER ..................................... 48  
  5.5 Summary ........................................................ 54  
6 Evaluation of DEPRECIATION WATCHER ........................... 55  
  6.1 Selection of Code Snippets ................................. 55  
  6.2 Procedure ....................................................... 57  
  6.3 Results ........................................................... 58  
    6.3.1 How Well Does DEPRECIATION WATCHER Perform? ........ 58  
    6.3.2 What Factors Affect How Well DEPRECIATION WATCHER Performs? ... 59  
  6.4 Evaluating Alternative Heuristics ....................... 61  
  6.5 Summary ........................................................ 63  
7 Discussion ........................................................... 65
List of Tables

4.1 Five systems that only deprecated APIs in one version. ......................... 28
4.2 Four systems removed all their deprecated APIs. ......................... 35
4.3 The versions for which each system removed their deprecated APIs. ....... 38

6.1 Deprecation Watcher has a precision of 1 and a recall of 0.86, using a conservative strategy for the API matcher. ......................... 59
6.2 Three different combinations of heuristics were evaluated on the same posts. The third one in the table performs better than the one we used to evaluate Deprecation Watcher. ......................... 61

A.1 Systems used in study. ............................................. 78
B.1 The number of APIs that have been removed, deprecated, and both deprecated and removed. ............................................. 79
C.2 Classification of deprecated messages. ............................................. 81
D.1 Three systems resurrected and deprecated some of their APIs after removing them. ............................................. 82
E.1 More deprecated Android APIs can be uniquely identified by using more pieces of information of an API but the usefulness of different pieces of information differ. ............................................. 83
H.1 Questions used in evaluation. ............................................. 88
## List of Figures and Illustrations

2.1 The accepted answer that seems to be able to address the developer's need. . . . 13

4.1 Our approach consists of two steps: data collection and analysis. . . . . . . . 27
4.2 Percentage of deprecation messages referring to concrete replacements in de-
scending order. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.3 Percentages of deprecation messages offering the rationale for the deprecation
of its APIs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.4 Percentages of deprecation messages with a concrete API removal plan. . . . 33

5.1 The structure of our framework. . . . . . . . . . . . . . . . . . . . . . . . . 41
5.2 A source code example on STACK OVERFLOW with code syntax highlighting. 42
5.3 The original HTML source for the source code example in Figure 5.2. . . . . 42
5.4 The HTML source for the source code example in Figure 5.2 after it is mod-
fied by a JAVASCRIPT code highlighting library. . . . . . . . . . . . . . . . . . 43
5.5 A profile can be constructed for code element `setContentView()` and then, based
on the profile, it is able to map this code element to a concrete API in the
ANDROID API. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
5.6 A sample code element profile for the example code element. . . . . . . . . . 45
5.7 A code example on STACK OVERFLOW. . . . . . . . . . . . . . . . . . . . 49
5.8 DEPRECIATION WATCHER identified the usage of a deprecated API in the
code example and appended a warning message below it. . . . . . . . . . . . 50
5.9 Algorithm to extract code elements and construct profiles for them. . . . . 52
5.10 Feedback that DEPRECIATION WATCHER presents to the user. . . . . . . 53

6.1 An example of chained method API calls. . . . . . . . . . . . . . . . . . . . 60
6.2 A code snippet with typecasting. . . . . . . . . . . . . . . . . . . . . . . . . 61
6.3 The return type helped detect the usage of a deprecated API `create` in line 3 62
6.4 `setTexture` was incorrectly detected as a usage of a deprecated API . . . . 63

F.1 A regular expression that matches variable declarations and definitions. . . 84
F.2 A regular expression that matches method and constructor names. . . . . 84
F.3 A regular expression that matches method calls. . . . . . . . . . . . . . . . . 84
Chapter 1

Introduction

API deprecation is an important tool used by developers of frameworks and libraries to evolve their systems because it temporarily maintains backward compatibility while urging users of the frameworks and libraries to transition to better APIs.

There have been anecdotal observations about how API deprecation is used in the Java standard libraries and a few third party libraries. For example, Henkel and Diwan [2005] observed that Java never removed its deprecated APIs and a few Java third party libraries removed their deprecated APIs more aggressively. However, the observations are based on a very small number of libraries and usually only a few versions of them, thus not necessarily generalizing to the wilder world. The observations also exclusively focus on one aspect of API deprecation in these systems; that is, how often deprecated APIs get removed. No systematic study has been done on how API deprecation is being used to evolve software frameworks and libraries. The first goal of this thesis is to address this question, by systematically studying API deprecation in 26 open source Java third party frameworks and libraries.

The second goal of this thesis involves usages of deprecated APIs in source code examples on the web. Developers make use of source code examples to learn how to use application programming interfaces (APIs) in a framework or library to solve a specific task. However, the official documentation of frameworks and libraries often lacks sufficient useful code examples. As a result, developers resort to online resources including Q&A sites like Stack Overflow1 and programming blogs that often offer source code examples demonstrating how to use APIs to accomplish a task. Despite being helpful, unlike official documentation, these online resources are separated from the development process of the frameworks and libraries and are not maintained by developers of the frameworks and libraries. When im-

1http://stackoverflow.com/
portant API changes are made, the code examples in resources on the web may not change accordingly. Consequently, source code examples on the web may contain APIs that have long been deprecated in a framework or library. Unaware of such information, developers may waste time in looking into the source code examples. Robillard [2009] showed that developers are indeed concerned with the timeliness of the code examples they find on the web. As far as we know no research has been done to address this problem. Our second goal in this thesis is to address one aspect of this problem by detecting deprecated API usages in code examples on the web.

1.1 Background and Motivation

Nowadays, most software systems use APIs provided by a number of frameworks and libraries to save time and effort. APIs are the exposed interfaces of underlying software components that are intended to be reused by developers. They effectively separate the complexity of how a software component is implemented from the complexity of how it is used. As a result, they allow developers to reuse functionalities without being concerned with their implementation details.

APIs are building blocks of software systems and it is important for both framework and library developers and users to agree on the set of APIs that a framework or library provides. In JAVA and many other object oriented languages, this is achieved by visibility-limiting features. Public and protected entities are considered APIs exposed by the library to users, except in a few special cases. For example, packages that are designated “internal” in frameworks or libraries are not considered exposed APIs despite the fact that they can be accessed and used in client applications.

Ideally, frameworks and libraries should keep their existing APIs unchanged and only add new APIs or change the inner implementation of existing APIs. In this way, applications that use their APIs will not be affected by the changes they introduce. However, in reality,
since they are software systems, frameworks and libraries evolve over time. The ISO/IEC standard for software evolution [ISO, 2006] extends the work by Lientz and Swanson [1980] classifying the changes into four categories including corrective, adaptive, perfective, and preventive changes. For frameworks and libraries, many of the changes manifest in the form of API deprecation. API deprecation is used by developers of frameworks and libraries as a way to retire an API, often in favour of a better alternative [How and When To Deprecate APIs].

API deprecation has been used in many situations and its instances can also be classified into the aforementioned four categories. Designing APIs for a system is known to be a challenging task [Henning, 2007, Tulach, 2008]. Some APIs are deprecated in order to correct a bad design decision made in earlier versions of a frameworks or library. Take the class java.util.Date in JAVA as an example: its initial design was flawed and many methods in this class were later deprecated in favour of a newly added class java.util.Calendar. Frameworks and libraries also deprecate their APIs to adapt to the changes of their dependencies. For example, when JAVA introduced enum types, making it a reserved word in version 5.0, several JAVA libraries had to deprecate some APIs in order to retire their own implementation of enum later. There are also more trivial reasons for the deprecation of APIs, such as spelling errors in the name of the API. For example, in the JAVA API, the getMaxDecent() method in class java.awt.FontMetrics was deprecated in favour of getMaxDescent(). In the third party JAVA library jackson-core, the API method isUnknownVersion() in class com.fasterxml.jackson.core.Version was deprecated in favour of the correctly spelt alternative isUnknownVersion().

The JAVA official documentation [How and When To Deprecate APIs] explains the meaning of deprecation as follows:

> You may have heard the term, “self-deprecating humor,” or humor that minimizes the speaker’s importance. A deprecated class or method is like that. It is no longer important. It is so unim-
important, in fact, that you should no longer use it, since it has been superseded and may cease to exist in the future.

Java provides a way to express deprecation because, as a class evolves, its API (application programming interface) inevitably changes: methods are renamed for consistency, new and better methods are added, and fields change. But such changes introduce a problem. You need to keep the old API around until developers make the transition to the new one, but you don’t want them to continue programming to the old API.

The same document also lists several valid reasons to deprecate an API, including

- it’s insecure, buggy, or highly inefficient
- it’s going away in a future release
- it encourages bad coding practices

Frameworks and libraries use API deprecation to keep old APIs around long enough that developers can migrate to new ones. They use deprecation messages to help developers to transition to new APIs. Deprecation messages are important for developers to make decisions about what to do with depreciated APIs and when to do it. According to the Java documentation, a good deprecation message should fulfill the following criteria. First, it should refer to the new API that replaces the deprecated API, if there is any. Second, it should help API users decide when to move to the new API. Lastly, it should briefly mention the reason for the deprecation. Previous studies [Ko et al., 2014, Robbes et al., 2012] on API deprecation messages only focused on the first criterion.

The question of when to remove an API naturally arises when it gets deprecated. Removing deprecated APIs too soon will break client code if users have not migrated their code. Removing APIs is a kind of breaking API changes: client code that compiled correctly
with the previous versions of the framework or library will fail to compile with the new version of the framework or library. A previous study [Henkel and Diwan, 2005] observed that the Java standard libraries never removed their deprecated APIs to preserve backward compatibility. On the contrary, there have been cases where deprecated APIs became undeprecated in Java. For example, Java deprecated `System.getenv(String)` in JDK 1.2 but un-deprecated it in JDK 5 because too many developers requested to reinstate this API. It reflects an underlying problem with deprecation: when developers make the decision about API deprecation, it is hard to know how many systems will be affected and how much effort will be required to transition to new APIs. As a result, many frameworks and libraries hold a conservative view about removing their deprecated APIs.

However, how common the conservative attitude towards removing deprecated APIs is among frameworks and libraries is unknown since it also has been observed that some Java third party libraries deprecate APIs more often than Java standard libraries and also remove their deprecated APIs more aggressively [Henkel and Diwan, 2005]. Aside from anecdotes about a few systems, no study has been systematically done to investigate how frameworks and libraries in general deal with their deprecations.

The approach that a framework or library takes on deprecations affects both the framework or library and its users. First, if a framework or library never removes any of its deprecated APIs, it will become harder to maintain over time. Second, the framework or system will also become harder to use for users. In many cases users have to first spend time investigating the deprecated APIs. For example, numerous questions have been asked on Stack Overflow about why most methods in the Java Date library have been deprecated and what to use instead. In many cases, users have to spend time understanding the inner implementation of APIs to understand why they are deprecated, which fails the purpose of APIs in the first place. Third, users may form the assumption that deprecated APIs will never be removed from the framework or library and continue using deprecated APIs despite
knowing it is not a good practice, which in turn makes it harder for the framework or library to remove the deprecated APIs. On the other hand, if a framework or library removes its deprecations too aggressively, it might prevent users from updating to its new versions.

In contrast to the conservative stance of some frameworks and libraries on removing deprecated APIs, some researchers [Kapur et al., 2010, Raemaekers et al., 2014] have found in some frameworks and libraries that APIs got removed without being deprecated in the first place. However, no research has investigated the relationship between the two. Do they co-exist in the same framework or library? Is API deprecation underused by frameworks and libraries?

Indeed, there are many questions about API deprecation left unanswered. How well are deprecation messages used in frameworks and libraries? Does un-deprecation happen often? How often do frameworks and libraries remove their deprecated APIs? When do they remove their deprecated APIs? These questions help us better understand how API deprecation has been used in frameworks and libraries. This knowledge can benefit both developers and users of frameworks and libraries. Part of our work in this thesis aims to answer some of these questions.

API deprecation has yet another effect on developers. When an API is deprecated, it is assumed that developers who use the framework or library will be aware of it. It might be the case when users exclusively consult the official documentation of the framework or library. However, the official documentation of a framework or library is no longer the only or even main way for developers to learn about how to use APIs. Indeed, according to a 2010 survey of over 3000 MSDN developers [Parnin et al., 2012], developers indicate that they learn about new APIs primarily through web search instead of using official documentation directly. Web searches often take a developer to websites like Stack Overflow or software development blogs. Parnin et al. [2012] showed that software development blogs and Q&A websites like Stack Overflow have a high coverage of API for many frameworks. Those
sites often offer concrete code examples on how to use APIs in the frameworks and are able to complement official documentation.

This decentralization of API documentation poses a great challenge to the synchronization of important API changes. Ideally, when an API is deprecated in a framework or library, the official documentation would immediately do the following to reflect the change. First, the documentation of that API should reflect that the API has been deprecated. Second, the usage of this API in all code examples in the documentation should be replaced or eliminated. In reality, however, the second part is hard to achieve. In the official ANDROID documentation, for example, the method `startManagingCursor(Cursor)` in class `android.app.ListActivity` has been deprecated since API level 11 yet it is still used in a source code example in the documentation about the class as of API level 21.\(^2\)

Therefore, it is even more unrealistic for unofficial documentation of deprecated APIs on the web and code examples referencing them to change immediately since they are disconnected from the official documentation. For example, on STACK OVERFLOW, answers are written to solve a question posted by someone and they are usually not actively maintained by the authors once the question is marked as solved. Therefore, there is no guarantee that the APIs used in answers will be up-to-date with the official API documentation. But at the same time, the questions and answers on STACK OVERFLOW are being actively viewed after they are marked solved by developers who may use a different version of the framework or library. In an active community that encourages everyone to contribute, like STACK OVERFLOW, updating one’s answers or even others’ to keep them up-to-date does happen but it cannot be relied upon. Deprecated API usages in the code examples may never be updated or even acknowledged. This is detrimental to developers. Developers may waste time in learning how to use deprecated APIs not knowing that they were deprecated. Granted, deprecated APIs are useful in certain cases. In ANDROID development for example, a developer may have to use deprecated APIs if the application is targeted for a device using an older

\(^2\)http://developer.android.com/reference/android/app/ListActivity.html
version of the Android operating system. But in all cases, it is vital to make developers aware of the usages of deprecated APIs so that they can make more informed decisions.

Previous studies [Dagenais and Robillard, 2012, Subramanian et al., 2014] have tried to map code elements in a source code example and code-like terms in informal documents to concrete APIs in a framework or library. Subramanian et al. [2014] also try to bridge the official documentation and code examples on the web by enabling them to cross reference each other. They have the potential to be adapted to detect deprecated APIs since both have the process of matching a code element in a code example to a concrete API. However, neither approach was built with API evolution in mind since they solely match code elements in a code example against the APIs from one version of a framework or library, whereas in reality, code examples can be written with APIs from different versions of a framework or library and at the same time developers may use different versions of a framework or library. As a result, these approaches are hard to be adapted to solve our problem. In this thesis we aim to detect deprecated APIs in source code examples without making two assumptions: the version of an API a code example on the web is written with and the version of the same API that a developer who resorts to the code example uses.

1.2 Research Questions and Objectives

Our work has two main research objectives. The first one is to answer the following research questions on API deprecation by empirically studying how API deprecation is being used in Java third party frameworks and libraries.

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?
Our second objective deals with a practical problem involving deprecated API usages on the web. We aim to detect deprecated API usages in source code examples on the web so developers can make informed decisions about the code examples they encounter on the web.

1.3 Contributions

This thesis has two main contributions.

• We conducted an empirical study on the practice of API deprecation in 26 open source Java third party frameworks and libraries. As far as we know, no study that focuses on API deprecation on this scale has been conducted. We answered our five research questions regarding how API deprecation has been used. Our answers can benefit both developers and users of frameworks and libraries in the long run.

• We proposed a novel framework to detect deprecated API usages in source code examples on the web. We implemented the framework as a prototype tool called Deprecation Watcher to specifically detect deprecated Android API usages in code examples on Stack Overflow. Our evaluation of tool shows the framework we proposed is promising.

1.4 Thesis Structure

This rest of the thesis is structured as follows.

Chapter 2 presents a motivational scenario for our second research goal, which is, detecting usages of deprecated API on the web.

Chapter 3 discusses related work on API deprecation and code examples on the web and why previous research does not address the problems we set out to tackle.

Chapter 4 presents our empirical study on API deprecation through tracking how API deprecation occurred and changed over time in 26 open source Java third party frameworks.
and libraries.

In Chapter 5 we propose a version sensitive framework to identify deprecated APIs in source code examples on the web and present our prototype tool Deprecation Watcher.

Chapter 6 presents our evaluation of Deprecation Watcher.

Chapter 7 presents a discussion about our work and its implications.

Chapter 8 discusses threats to the validity of our work. It discusses both the threats to the validity of our empirical study on API deprecation and our evaluation of Deprecation Watcher.

Chapter 9 concludes the thesis.
Chapter 2

Motivation

This chapter presents the motivation for detecting deprecated API usages in code examples on the web.

Nowadays developers increasingly resort to web search for help because it leads them to useful sites like STACK OVERFLOW and various technical blogs, many of which contain helpful code examples. However, the code examples are written with a specific version of a framework or library in mind and APIs used in these code examples can get deprecated in new versions of the system. This can be a problem for developers since they may not know that before devoting time into the code examples. As a result, they can waste time and energy in such code examples.

In Section 2.1 we present a motivational scenario to demonstrate in detail how deprecated API usages in source code examples on the web can waste a developer’s time and energy and how a tool would help.

2.1 Motivational Scenario

Imagine an ANDROID developer who needs help on how to capture a picture from a webview on an ANDROID device. Like a lot of developers, instead of searching in the official ANDROID documentation, he decides to look for help on the web through GOOGLE. Typical search keywords for his needs would be “android capture picture webview”. So he switches from his work environment (ANDROID STUDIO) to a web browser and uses those keywords to start a GOOGLE search. The top four results for this query returned by GOOGLE are all from STACK OVERFLOW. Suppose the developer clicks the first link.\(^1\) The developer sees

\(^1\)http://stackoverflow.com/questions/7702565/capture-picture-from-android-webview
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.² It seems like the accepted answer would address this developer’s problem too so the developer decides to look into the answer. The answer consists mainly of two code blocks with little explanation but the code blocks seem quite self-explanatory. The screenshot of the answer is shown in Figure 2.1.

After spending a few minutes understanding the code, the developer starts to think about integrating the code into his own work. The code blocks also seem self-contained and do not have dependences other than that the first code block uses the function defined in the second code block. Therefore the developer is 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: methods `setPictureListener()` and `capturePicture()` in class `android.webkit.WebView` and `onNewPicture()` in class `android.webkit.WebView.PictureListener`. Methods `setPictureListener()` and `onNewPicture()` were deprecated in **Android** API level 12 and method `capturePicture()` was deprecated 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 documentation³ for method `setPictureListener()` merely states “This method was deprecated in API level 12. This method is now obsolete.” So it is still unclear what to use instead.

He again decides to use **Google** search to see how others might have solved this problem.

---

² All data taken from the webpage as of September 21, 2015.
³ WebView — **Android** Developers
public void onPageFinished(WebView webview, String url) {
    Picture picture = webview.capturePicture();
}

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;
}

Figure 2.1: The accepted answer that seems to be able to address the developer’s need.
So he uses keywords “android setpicturelistener obsolete” to do another search. However, this time the top result from the Google search is a Stack Overflow post⁴ that merely refers back to the official documentation.

Having wasted so much time, the developer decides to undo the changes he has made to his own code and go back to his first Google search instead of looking for the replacements to the deprecated APIs. This time he clicks the second link in the Google search results.⁵ 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 manually evaluating the code example again, he decides to try using this code example. And finally this code example addresses his problem.

At this point, the developer has wasted a significant amount of time and energy doing the following:

- Understanding the first code example without realizing the code example uses deprecated APIs.
- Trying to integrate the code example into his own code.
- Looking for replacements for the deprecated APIs.
- Undoing changes to his code and looking for new examples.

On the other hand, if the developer is able to know whether a code example contains usages of deprecated APIs and furthermore, which APIs are deprecated, he can either avoid spending time understanding the code example and start searching for a new code example immediately or begin looking for the replacements of the deprecated APIs early. Only after that, he may try to integrate the code example into his own code.


2.2 Summary

This chapter described a motivational scenario for detecting deprecated API usages in code examples on the web in detail.

The motivational scenario demonstrates how not knowing whether a source code example on the web contains usages of deprecated APIs can greatly waste a developer’s time and energy. The developer first spends the time examining the code example. He may also attempt to integrate it into his code base and have to undo the changes later. After realizing the existence of deprecated APIs in the code example, he then need to either search for the replacements of the deprecated APIs or search for new code examples. Looking for the replacements of deprecated APIs can be time-consuming in many cases, for example, when the official documentation does not offer useful information or no replacements for the deprecated APIs exist. If he decides to search for another code example instead, he may once again encounter a code example using deprecated APIs. On the other hand, with the knowledge of whether a code example contains deprecated APIs before investing time into it, a developer can avoid wasting time and energy in unfruitful activities.
Chapter 3

Related Work

In this chapter we discuss previous work on API deprecation and source code examples on the web.

API deprecation is a manifestation of software evolution; therefore, our work is closely related to the broad research area of software evolution. But more concretely our work belongs to its subdomain of API evolution. In this area, two types of research have been done. The first type of research concerns the nature of API evolution and its effects. Dig and Johnson [2005] found that over 80% of breaking API changes are due to code refactorings, although Cossette and Walker [2012] call this result into question. Much research [Linares-Vásquez et al., 2014, McDonnell et al., 2013, Robbes et al., 2012] has been done on how developers react to API evolution. The second type of work focuses on how to better deal with API evolution. Numerous approaches [Chow and Notkin, 1996, Dig et al., 2008, Henkel and Diwan, 2005, Nita and Notkin, 2010, Perkins, 2005, Xing and Stroulia, 2007] have been proposed to help developers migrate broken APIs due to breaking API changes. Some of the approaches require the participation from developers of frameworks and libraries while others do not. To date, the work in API evolution mostly focuses on breaking API changes while API deprecation is greatly understudied. For instance, only one approach [Štrobl and Troníček, 2013] has been proposed to specifically migrate deprecated APIs. Granted, the approaches proposed for migrating removed APIs might be able to work for deprecated APIs as well, but the sharp contrast is still an indication that API deprecation has been overlooked in the research community. We hypothesize that there are two main reasons behind this. First, many assume API deprecation eventually leads to breaking API changes due to the removal of deprecated APIs. Therefore, techniques targeted at breaking API changes will
eventually be used. However, no formal empirical study has confirmed that deprecated APIs will eventually be removed and there are anecdotal observations that suggest otherwise. Second, since API deprecation does not break code immediately, many tend to focus on breaking API changes that cause immediate problems.

One problem caused by API deprecation is that deprecated APIs exist in source code examples on the web; since the APIs in the code examples are not directly linked to the concrete APIs of a framework or library, developers are not able to discover them quickly if at all. Recently, two main approaches [Dagenais and Robillard, 2012, Subramanian et al., 2014] have been proposed to link all APIs in code examples on the web to concrete APIs. These approaches are not sufficient to address our problem in the context of API evolution where code examples on the web can be written with different versions of a framework or library and developers can also be using different versions of the framework or library. Our work addresses this problem by proposing a version sensitive framework to specifically match code elements in code examples to concrete, deprecated APIs in a framework or library.

In Section 3.1 we discuss previous research in API deprecation and in Section 3.2 we discuss previous research related to code examples on the web.

3.1 API Deprecation

Robbes et al. [2012] studied the ripple effect of API deprecation in a SMALLTALK ecosystem and how developers react to API deprecation. They found that deprecation messages can not be relied upon to be helpful because they can be missing or not offer concrete advice. They only categorize deprecation messages based on whether they offer replacements and whether the suggestions are followed by developers; therefore, they do not present a complete picture of how deprecation messages are used. Their study is done on only two SMALLTALK systems and might not be representative of how framework and library 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 in the same system. Ko et al. [2014] studied the quality of deprecation messages in eight Java libraries and found that only 61% of the deprecation messages provide replacement APIs, while rationales and concrete examples on how to use new APIs are rarely given. Different from their study, 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 larger scope than theirs.

Hou and Yao [2011] conducted a case study of the evolution of the AWT/Swing APIs to classify the intent behind API deprecations and API additions based on deprecation messages and other resources. Their study focuses on why APIs get deprecated in the framework while our work focus on how API deprecation is used. Linares-Vásquez et al. [2014] 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. [2014] 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.

Much of the research related to API deprecation is problematic so far. Raemaekers et al. [2014] studied deprecation patterns in Java systems on Maven Central Repository.
They found that APIs that were marked as deprecated were never removed. However, in their study they only considered an API to be deprecated if it has a @Deprecated annotation. Their approach is problematic because the @Deprecated annotation was not introduced until JDK 5.0. APIs were deprecated via the @deprecated Javadoc tag (available since JDK 1.1) before that and APIs can still be deprecated only with the Javadoc tag. Therefore, the credibility of their result is questionable. On the other hand, many researchers [Dig and Johnson, 2005, Taneja et al., 2007, Xie et al., 2008] 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 many frameworks and libraries have not been removed despite having remained as deprecated for years; there has been little 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. How often does the deprecate–replace–remove cycle actually occur? How often does un-deprecation happen? No research has addressed these questions yet.

Our research addresses some of the unanswered questions through tracking how API deprecation occurs and how deprecated APIs change over time.

3.2 Source Code Examples on the Web

Code examples are helpful for developers to learn APIs [Robillard, 2009]. Code search engines like Google Code Search\(^1\), Krugle\(^2\), and Open Hub\(^3\) are able to search code from a vast number of open source projects that are available online. However, these code search engines are limited because they are mostly keyword-based [Reiss, 2009]. Several code search techniques [Bajracharya et al., 2006, Holmes and Murphy, 2005, Reiss, 2009] have been

---

\(^1\)Google Code Search engines was shut down in March 2013

\(^2\)http://www.krugle.com/

\(^3\)http://code.openhub.net/, merged from KODERS and OHLOH CODE.
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 more than timeliness.

Two approaches [Dagenais and Robillard, 2012, Subramanian et al., 2014] are related to our work because they both aim to link code elements in a code snippet to concrete API in a framework or library. **RecoDoc** [Dagenais and Robillard, 2012] was proposed to link code-like terms in learning resources of a framework or library to concrete API elements in the framework or library. It aims to solve the problem that the learning resources are separated from concrete APIs and therefore may not be able to reflect API changes. The **Baker** tool [Subramanian et al., 2014], on the other hand, 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 an concrete API in a framework or library. To uniquely identify an API, a fully qualified name (FQN) is required while names in free text and code snippets tend to be ambiguous. Dagenais and Robillard [2012] found that direct name matching does not work for most methods because of name collisions. Methods with the same name can be found to be declared in more than 13 different types on average. **RecoDoc** uses a complex 5-layer filtering pipeline to conquer the challenge of the ambiguity while **Baker** uses a more complicated Oracle.

Although both approaches work well for their own purposes, neither of them is able to address the problem of detecting deprecated API usages in source code examples on the web. The reason is that they have underlying assumptions about code examples that do not remain valid in the context of API evolution. For example, **Baker** has an underlying assumption 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 problem 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. For example,
McDonnell et al. [2013] showed that the Android API evolves at the rate of 115 API updates
per month on average, many of which are deprecations or deletions. 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. Instead of trying to
adapt these approaches and significantly change how they work, which would take too much
effort, we propose a new approach. Unlike RecoDoc and Baker, our approach is designed
to work in the context of API evolution and does not assume that code examples on the
web are written with the a specific version of a framework or library. Our work also differs
from RecoDoc and Baker in that our approach only aims to identify deprecated APIs in
code examples instead of mapping every code element to concrete APIs, thus allowing our
approach to be much more straightforward and easier to implement.

3.3 Summary

In this chapter, we discussed how our work relates to previous research around API depre-
cation and code examples on the web.

Our work differs from current research in API deprecation in two ways. First, we set
out to answer several important questions not answered by previous research. Second, our
research is more extensive. For instance, our work on the quality of deprecation messages is
more extensive than the previous work [Ko et al., 2014] in this area.

Some approaches [Dagenais and Robillard, 2012, Subramanian et al., 2014] aim to uniquely
map code elements in the code examples on the web to concrete APIs. However, these ap-
proaches are lacking in the context of API evolution because they have underlying assump-
tions about the versions of the frameworks or libraries used in the code examples on the web. In contrast, our approach does not make such assumptions and is designed to work in the context of API evolution. Our approach also only aims to detect deprecated API usages instead of mapping every code element in a code example to a concrete API.
Chapter 4

Data Collection and Analysis

In this chapter, we set out to address our five research questions on API deprecation. Our goal is to empirically study how API deprecation has been used in Java third party frameworks and libraries. In particular, we focused 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). In this and the following chapters, deprecated APIs (or deprecations) exclusively refer to deprecated API methods and constructors unless otherwise specified.

Section 4.1 describes the criteria we used to choose the systems used in the study. Section 4.2 describes the process we used to gather API deprecation data. And lastly, Section 4.3 describes the analysis we did on the data.

4.1 Selection of Frameworks and Libraries

We used MVNREPOSITORY\(^1\) to select libraries used in this study. MVNREPOSITORY is a search engine for Maven projects on Maven Central Repository.\(^2\) It also ranks how popular those projects are by how many other projects on Maven Central Repository use them. However, not all of the projects listed are Java third party frameworks and libraries. For example, some projects are JVM languages like Scala and Groovy and some projects are modules such as Logback-Core. We only selected projects that are Java third party frameworks or libraries.

Our selection of frameworks and libraries was based on the following criteria. First, we chose open source frameworks and libraries because their source and binary code are

\(^1\)http://mvnrepository.com
\(^2\)http://central.sonatype.org/
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 an deprecated API gets removed. The rationale for using source code and binary code of the frameworks and libraries to get API deprecations and removals is detailed in the next section. 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.\textsuperscript{3} The ranking is based on how many other projects on MAVEN CENTRAL REPOSITORY depend on it. Third, we chose frameworks and libraries that have been around for at least 6 years.

In the end we selected 26 Java open source third party frameworks and libraries for our study. The list of frameworks and libraries is in Appendix A. The average number of releases for the systems in our study is 27 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.

4.2 Gathering Data

To track deprecated APIs, we need to obtain the following information about API deprecation in the chosen frameworks and libraries.

- When exactly did an API get deprecated

- What happened to a deprecated API method or constructor after being deprecated

We chose to use source code and binary code of the frameworks and libraries to answer these questions because they are the most reliable sources of API changes. We also used tools to extract API deprecations and API changes instead of doing it manually.

\textsuperscript{3}http://mvnrepository.com/popular, accessed in March 2015
We considered several other approaches to achieving that. First, deprecated APIs are usually listed in a library’s official documentation. For example, the online documentation for version 1.2.17 of LOG4J\(^4\) has a list of deprecated classes, fields, methods, and constructors. As for obtaining removed deprecated APIs, one can go through the release notes and changelogs of every release of a library manually to check for deleted APIs and try to match them against previous deprecated APIs.

There are several problems with the aforementioned approaches. First, the whole process is manual, thus time-consuming and error prone. Second, official documentation for old versions of a library can be hard to locate. Many frameworks and libraries use a single set of documentation for multiple versions and the documentation only reflects the latest version of the library. For instance, the 1.2.x version of LOG4J now shares just one online documentation. Therefore, it can be hard to know when exactly an API got deprecated even if it is listed in such documentation. Some deprecation messages do specify this information but such cases are not the norm. Release notes and changelogs, on the other hand, include too much information unrelated to the deletion of deprecated APIs. Furthermore, release notes also might not be comprehensive about the API changes made in a release because they are written manually by developers.

Our process of gathering deprecation data was as follows. We first tried to download as many versions of the libraries as possible from their official website if an official archive was available there. We have found that many libraries, especially libraries released by the Apache Software Foundation\(^5\), have well maintained archives of previously released versions of the libraries. When an official archive was not available, we downloaded the source and binary code of the libraries from the MAVEN CENTRAL REPOSITORY\(^6\) instead.

\(^4\)https://logging.apache.org/log4j/1.2/apidocs/
\(^5\)http://www.apache.org/
\(^6\)http://central.sonatype.org/
After obtaining the source and binary code of the libraries, we use two tools to process them. We used QDox\textsuperscript{7} to parse the source code of every version of the libraries to get a list of deprecated APIs in each version. QDox is a parsing tool that can parse Java code and extract various information about the code entities in the code. We used it to get deprecated APIs from a library by parsing the annotations and Javadoc of an entity. We use Clirr\textsuperscript{8} to get important API changes between two consecutive versions of a library. Clirr checks Java libraries for binary and source compatibility between a new version of the library and a previous one. It is also being used by several projects in Apache Software Foundation to check API changes in new versions of the libraries. We only used it to obtain API removals since we are only concerned with the deletion of deprecated APIs. Our approach is illustrated in Figure 4.1.

4.3 Data Analysis

In this section, we use API deprecation and API removal data to answer the research questions we put forward in Chapter 1.

RQ1. Is API deprecation underused?

We found that all of the systems in our study deprecated some APIs during the time period examined in our study. This shows that using API deprecation to evolve APIs in Java third party frameworks and libraries is a common practice.

However, we also found that five systems only deprecated APIs in one version. The number of deprecated APIs in these versions is also small. These systems are listed in Table 4.1.

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

\textsuperscript{7}qdox.codehaus.org
\textsuperscript{8}http://clirr.sourceforge.net/
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 releases over the course of 12 years, which is in sharp contrast with the number of its deprecations.

As a matter of fact, in the majority of the systems, removed APIs outnumber deprecated APIs significantly and in many cases the two sets do not overlap. Many APIs were removed without being deprecated first. See Appendix B for the detailed comparison of the number of removed APIs and the number of deprecated APIs for all the systems.

For example in commons-collections 4.0, besides removing deprecated APIs, it also “removed features now supported by the JDK or other Apache Commons libraries” without
deprecating relevant APIs first, as stated in its release notes.9

We found that API deprecation is indeed underused since many APIs were removed without being deprecated first.

**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 come with a message but we found that the majority of them do. 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. The result is shown in the table in Appendix C.

**RQ2.1 How often do deprecation messages refer to a concrete replacement?**

We manually classified deprecation messages into four categories based on whether a concrete

---

9[http://commons.apache.org/proper/commons-collections/release_4_0.html](http://commons.apache.org/proper/commons-collections/release_4_0.html)
A replacement is given: \textit{without replacement, no mention of a replacement, a vague replacement,} and \textit{a concrete replacement}. The difference between \textit{without replacement} and \textit{no mention of a replacement} is that the deprecation message in the former specifically states that there is no replacement for the deprecated API while when a replacement is not mentioned in a deprecation message, it is not certain whether one exists or not. \textit{A vague replacement} only offers the general directly for migrating without referring to a concrete API.

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.

Figure 4.2 shows the percentages of deprecated messages which pointed out a concrete replacement for all systems.

\textbf{RQ2.2 How often do deprecation messages offer explanations?}

Knowing why a certain API has been deprecated helps developers make informed decisions. If an API was deprecated because of security reasons, users are more likely to replace it. Deprecation messages were classified into two categories based on this question: \textit{containing an explanation} and \textit{containing no 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\%. Figure 4.3 shows the percentages of deprecation messages offering the rationale for the deprecation of its APIs in descending order.

Our result shows that developers of these systems tend not to explain their decisions
Figure 4.2: Percentage of deprecation messages referring to concrete replacements in descending order.
Figure 4.3: Percentages of deprecation messages offering the rationale for the deprecation of its APIs in all systems sorted in descending order. It shows that developers tend not to communicate the rationale for their decisions in deprecation messages.
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.

**RQ2.3 How often do deprecation messages specify a time frame for deletion?**

When will an API be removed in the future affects the decisions developers need to make about their projects. If an API is expected to be removed soon, developers would assign a higher priority to the task of replacing that API. We classify API deprecation messages into three categories based on whether a plan to remove a deprecated API is in place: no removal plan, a vague removal plan, and a concrete removal plan. A vague removal plan merely states that the deprecated API would be removed in the future but does not specify when exactly, 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. Figure 4.4 shows the percentages of deprecation messages offering a concrete removal plan.

In summary, our results show that API developers do not make use of deprecation messages well enough to help API users to migrate deprecated APIs. Less than a quarter of the systems specified all replacements of the deprecated APIs. Worse still, most systems do not offer explanations for the deprecation of their APIs or specify when their deprecated APIs are expected to be removed.

**RQ3. How often does un-deprecation of APIs occur?**

Sometimes an API gets un-deprecated after being deprecated. This signals that developers of the framework or library no longer think it is possible to ask users to migrate to the new API. This could happen when they deprecate a popular API.

We tracked deprecated APIs in each library over time and see if they stopped being
Figure 4.4: Percentages of deprecation messages with a concrete API removal plan.
marked as deprecated in a later version. The result shows that almost half (12 in 26) of the systems have un-deprecated some of their deprecated APIs during the time span examined in our study. By manually examining these cases, we observed that in most cases the replacements still exist.

We also discovered a rather interesting phenomenon when trying to answer this research question. We found that in 3 systems, namely JUNIT, LOG4J, and XERCESIMPL, some APIs first got removed, then were brought back and marked as deprecated. (See Appendix D for a complete list.) We call this phenomenon remove–resurrect–deprecate in comparison to the widely acknowledged deprecate–replace–remove cycle of using deprecation to evolve APIs. We hypothesize that this indicates that library developers have falsely assumed that these APIs were not used by their users and went forward to remove them. For example, JUNIT has deleted two APIs before adding them back and deprecating them instead. It removed the protected method rules (java.lang.Object) in class org.junit.runners.BlockJUnit4ClassRunner in version 4.8.2 and added it back in JUNIT 4.9 after a user opened an issue about it on GitHub.\footnote{https://github.com/junit-team/junit/issues/248} It also removed the method exit (int) in class org.junit.internal.RealSystem in version 4.11-beta-1 but added it back in 4.12-beta-2 and marked it as deprecated. Internal packages are usually not considered as part of the API a library provides, and so they are subject to liberal changes. Indeed, the internal package org.junit.internal is not present in JUNIT’s official documentation.\footnote{http://junit.sourceforge.net/javadoc/} However, since internal packages are public and therefore available to users, developers may be forced to evolve them as if they were a part of the official API if they are widely used.

Un-deprecation and the phenomenon of remove–resurrect–deprecate indicate that developers of the systems lack the full knowledge of how their systems are being used.

**RQ4. How often are deprecated API removed later?**

We tracked deprecated APIs in the systems over consecutive versions and if they stop ap-
pearing in a version, they are deemed as removed. We found that 42.3% of the deprecated APIs got removed later. This number is in direct contrast to Raemaekers et al. [2014] where they found no deprecated APIs were removed. Two factors can explain this discrepancy. First, as pointed out in Chapter 1, they only obtained a subset of deprecated APIs as they only used the `@Deprecated` annotation to extract deprecated APIs. Second, on Maven Central Repository, a single framework or library can correspond to different artifacts. For instance, Commons Collections corresponds to two different artifacts: commons-collections (before version 4.0) and commons-collections4 (after version 4.0). Our work also calculated deprecated API removal across this boundary while their approach incorrectly ignored it. This number is also higher than what we expected because of our past experience of using Java and certain Java third party libraries.

Our 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 got removed. Only 15.4% (4 out of 26) of the systems removed all their deprecations. These systems and the number of deprecated APIs in them are shown in Table 4.2. Two of four systems only had one or two API deprecations in total.

<table>
<thead>
<tr>
<th>Library</th>
<th>Deprecated APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>commons-collections</td>
<td>10</td>
</tr>
<tr>
<td>commons-configuration</td>
<td>15</td>
</tr>
<tr>
<td>maven-model</td>
<td>2</td>
</tr>
<tr>
<td>nekoHTML</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.2: Four systems removed all their deprecated APIs.

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 spans in our study. For example, commons-io has not removed any of its deprecated API for as long as 8 years, during which it released 13 versions.
Some of the deprecations 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.

This sentiment was explicitly expressed by a maintainer of JUNIT in a GitHub issue\(^\text{12}\) dating back to June 3, 2013:

\[
I \text{ can’t seem to remember that we have ever actually deleted any API from JUNIT. Over the years, we have found better ways to do things, and use deprecation as a way to point users and extenders to the new, better ways.}
\]

\[
When I was a younger API maintainer, I used to believe that we would some day be able to delete the old ways, but I’ve learned that is rarely a win—slightly less work for the maintainers, much more work for the users.
\]

\[
As this API evolves, we will probably continue to mark methods and classes deprecated when better options become available. But, as I said, you can rest assured that you have very high odds that the extension points you use today will continue to work for years to come, even if they are someday marked deprecated because we’ve found something better.
\]

On the other hand, despite the fact that more than half of the systems never removed any of their deprecated APIs, more than 40% of the deprecated APIs were removed overall.

\(^\text{12}\)https://github.com/junit-team/junit/issues/689
RQ5. When do deprecated APIs get removed?

To answer the last research question, we found that less than half (12 in 26) of the systems ever removed their deprecated APIs, up to the present day. So when did these API removals occur? To answer this question, we chose to examine when deprecated APIs got removed in systems that have removed more than two deprecated APIs. We chose this condition 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 transition to major releases, and other versions. Different frameworks and libraries have different conventions about what can be considered as a major release even though a release that increases the major version number is considered as a major release. Specifically, in STRUTS the transition from 1.2 to 1.3 was considered to be a major release, and in LOG4J the transition from 1.1.3 to 1.2 was considered a major release as well.

As is shown in Table 4.3, 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 removed deprecations in two non-major releases: 1.1.3 and 1.2.1.

Therefore, 74.4% of the removed deprecated APIs were removed during the transition to major releases in the above systems and 99.0% of the removed deprecated APIs were removed either before version 1.0 or during transitions to major releases. This reflects that API developers are concerned with the backward compatibility of their systems and are intentional when introducing breaking API changes. This is surprising given that we also found that many APIs were removed without being deprecated first.
Table 4.3: The versions for which each system removed their deprecated APIs. “n/a” means the first version of that library used in our study is already version 1.0 or later. Five of the seven systems exclusively removed their deprecations either before version 1.0 or during a transition to a major release while two of the seven systems also removed deprecations in other versions.

<table>
<thead>
<tr>
<th>Library</th>
<th>Before version 1.0</th>
<th>Transition to major releases</th>
<th>Other versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMONS-COLLECTIONS</td>
<td>n/a</td>
<td>yes (10)</td>
<td>no</td>
</tr>
<tr>
<td>COMMONS-CONFIGURATION</td>
<td>n/a</td>
<td>yes (15)</td>
<td>no</td>
</tr>
<tr>
<td>COMMONS-LANG</td>
<td>n/a</td>
<td>yes (29)</td>
<td>no</td>
</tr>
<tr>
<td>JDOM</td>
<td>yes (69)</td>
<td>yes (14)</td>
<td>no</td>
</tr>
<tr>
<td>JODA-TIME</td>
<td>n/a</td>
<td>yes (18)</td>
<td>no</td>
</tr>
<tr>
<td>STRUTS</td>
<td>n/a</td>
<td>yes (65)</td>
<td>no</td>
</tr>
<tr>
<td>LOG4J</td>
<td>n/a</td>
<td>yes (82)</td>
<td>yes (3)</td>
</tr>
</tbody>
</table>

4.4 Summary

In this chapter, we introduced how we selected the systems used in our study, the process of gathering API deprecation and removal data from the selected systems and also answered the research questions on API deprecation we put forward in chapter one by analyzing the data. Our findings are as follows.

- API deprecation is significantly underused by developers of frameworks and libraries. Many APIs were removed without getting deprecated first.

- Deprecation messages are lacking on all three criteria we used: whether a replacement is given, whether a rationale is given, and whether an expected removal time frame is given.

- Un-deprecation happened in almost half of the systems we studied and interestingly the phenomenon *remove–resurrect–deprecate* of APIs occurred in three systems as well.

- Frameworks and libraries are reluctant to remove their deprecated APIs, as is evidenced by the fact that more than half of the systems never removed any of their deprecations.
• More than 40% of all deprecated APIs were removed and almost all of them were removed either before version 1.0 or during the transition to major releases.
Chapter 5

Model and Tool

In Chapter 2, we examined a motivational scenario in which a developer wastes a significant amount of time and energy examining code examples on the web without knowing there are deprecated API usages in the code examples beforehand. In this chapter, we aim to tackle the problem by detecting usages of deprecated APIs in code examples on the web. To solve this problem, we first need to obtain the code examples from a webpage, then we need to figure out whether an API in a code example is deprecated for a developer. However, we cannot make assumptions about either the version of the API the code example is written with, or the version of the API the developer is using.

We propose a version sensitive framework to solve this problem. The framework consists of three components: a code element extractor, a version sensitive API matcher, and a visual feedback component. The code element extractor and visual feedback component work in a web browser while the version sensitive API matcher works on a server. The code element extractor is used to extract source code examples from a webpage, identify code elements from the code examples, and construct a profile for each of the code elements. It uses a set of regular expressions along with the context in the code snippet to extract as much information about code elements in a code snippet as possible. It then sends this information to the API matcher. The version sensitive API matcher is used to match the profiles of code elements to deprecated APIs in a target framework or library. It uses a repository that contains all APIs of the target framework or library along with information about the states of the APIs in each version of the target framework or library. It matches the profiles to concrete APIs in the database using configurable heuristics. The visual feedback component then presents the feedback to the user if the server confirms that deprecated APIs have been matched.
The structure of the framework is shown in Figure 5.1.

Our framework can be easily extended, for example, to detect removed APIs as well but we chose to focus on API deprecation specifically in our research.

Sections 5.1, 5.2, and 5.3 below describe the code element extractor, the version sensitive API matcher and the visual feedback component respectively. In Section 5.4 we then introduce our prototype tool, named Deprecation Watcher, as a proof of concept. Deprecation Watcher is a straightforward implementation of our proposed framework.
5.1 Extracting Code Elements on the Web

Source code examples on the web are usually wrapped in the `<pre>` and `<code>` HTML tags to preserve the pre-formatted text and signify a code fragment. The `<code>` tag also makes it easy for syntax highlighting JavaScript libraries to detect the programming language in the code fragment. The libraries then can highlight keywords and code elements in the code example to make it easier to read.

Figure 5.2 shows an example of a source code example\(^1\) on STACK OVERFLOW. Figure 5.3 shows its original HTML source code. Figure 5.4 shows the HTML source code after it was modified by the JavaScript syntax highlighting library used by STACK OVERFLOW.

```java
Display display = getWindowManager().getDefaultDisplay();
Point size = new Point();
display.getSize(size);
int width = size.x;
int height = size.y;
```

Figure 5.2: A source code example on STACK OVERFLOW with code syntax highlighting.

```html
<pre><code>Display display = getWindowManager().getDefaultDisplay();
Point size = new Point();
display.getSize(size);
int width = size.x;
int height = size.y;
</code></pre>

Figure 5.3: The original HTML source for the source code example in Figure 5.2.

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 webpage they display available to extensions. The model we propose works as an extension to a web browser because it needs access to the webpage where the source code examples are.

\(^1\)http://stackoverflow.com/questions/1016896/get-screen-dimensions-in-pixels/
To extract code elements from the source code examples, we first extract the code snippets on a webpage using HTML tags. We also process code snippets and reconstruct the lines in the code snippets before extracting code elements from them. To match code elements in the code snippets, we created a set of regular expressions and match them to every line in the code snippet and construct a profile for them. The extracting of code elements is processed locally in the browser. Some approaches [Dagenais and Robillard, 2012, Subramanian et al., 2014] use a full-blown Java parser instead, parsing code snippets into their corresponding
abstract syntax trees (ASTs). This approach tends to work well by utilizing structural
information in a code snippet. It often requires code snippets to be sent to a server to be
parsed. However, the parser will not work properly in cases where the code snippets are
malformed.

```java
protected Dialog onCreateDialog(int id) {
    Dialog dialog;
    switch (id) {
    case 0:
        dialog = new Dialog(this);
        dialog.setContentView(R.layout.paused);
        dialog.setTitle("Game Paused");
        dialog.show();
        break;
    default:
        dialog = null;
    }
    return null;
}
```

Figure 5.5: A profile can be constructed for code element `setContentView()` and then, based
on the profile, it is able to map this code element to a concrete API in the Android API.

The next step is to construct a profile for every code element in the source code example.
Many pieces of information about a code element can be used in the profile. For example,
in the code example in Figure 5.5, for the code element `setContentView()`, the following types
of information can be extracted.

**Type of a code element.** The type of an API can be class, interface, method, constructor,
or field. The type of code element `setContentView` is method because it matches the regular
expression for a method call.

**Name of a code element.** The name of the example code element is `setContentView()`.

**Name of class.** The name of the enclosing class for the code element is `Dialog`. This
information is inferred through the context in the code. In the second line of the code
example, a variable named `dialog` of type `Dialog` is declared. This information is stored into
the context for the code example. In the 6th line, method call `setContentView(R.layout.paused)`
Figure 5.6: A sample code element profile for the example code element. This profile is then sent to the API matcher.

is invoked on dialog.

**Argument number.** The code element takes one argument. This information is also extracted through a regular expression.

**Return type.** The return type of the code element is `void`.

The profile for the code element `setContentView` constructed with the information above is shown in Figure 5.6.

The profile that a code element extractor builds for a code element depends on several factors. First, it depends on how much information can be extracted. Many code examples on the web lack information such as the fully qualified name of a class and as a result such information cannot be extracted. Second, it is closely related to the strategy that the API matcher uses to match a deprecated API since the API matcher uses the code element profile for matching.

5.2 Version Sensitive API Matcher

The goal of the version sensitive API matcher is to map a profile of a code element into a concrete deprecated API in the target framework or library. Our matcher 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 4.2. Our API matcher is version sensitive in that it returns its result based on the version the API matcher uses. This can be configured
by developers. Our API matcher needs to be version sensitive because an API can be in
different states in different versions of a framework or library. It may be absent in the earlier
versions of a library, to be added later. Moreover, it can also become deprecated and stay
deprecated for a few versions until it gets removed. For instance, the API `setPluginsEnabled`
in Android class `android.webkit.WebSettings` was deprecated in API level 9 and stayed as
deprecated until API level 18. If our API matcher is configured to use API level 16, for
example, the state of the API will be given as deprecated.

The API matcher makes use of a set of heuristics to match code element profiles to
deprecated APIs in the database. Heuristics are used because it is difficult to obtain the
fully qualified name of an API from a code example on the web directly. We can use pieces of
information about a code element that can be extracted from the code example to construct
heuristics to use instead. The basic heuristic can be that the name in the code element
profile has to match exactly with the name of a deprecated API. Another heuristic can
be the number of arguments that a code element takes. Even though more heuristics can
possibly lead to a better result, the best heuristics for matching deprecated APIs may differ
for different systems. For example, in a library where all the deprecated APIs have unique
names, matching the name is sufficient, whereas in a library where none of the deprecated
APIs has a unique name, naturally more heuristics are required. Therefore, when deciding
the set of heuristics to use for matching deprecated APIs in a target framework or library,
it is beneficial to explore the characteristics of the deprecated APIs in the framework or
library first. For instance, when designing the tool DEPRECATION WATCHER described
in Section 5.4, we conducted a preliminary study on all the deprecated ANDROID APIs to
estimate how a few different sets of heuristics may perform. Concretely, we used different
pieces of information of an API to see how well they can uniquely identify the deprecated
APIs from the set of all ANDROID APIs. Several observations from the study helped us
decide on the final combination of heuristics. First, matching names alone did poorly in
uniquely identifying a deprecated Android API because many different APIs have the same name. In fact, just 53.4% of the deprecated Android APIs can be uniquely identified solely by their names and on average for every deprecated Android API, there are 3.2 Android APIs with the same name that are not deprecated. Second, we found that using the enclosing type name of a deprecated API along with the name of the API was quite effective, able to identify 77.7% of the deprecated APIs. On the other hand, using the return type and the name of a deprecated Android was able to identify merely 58.5% of the deprecated APIs and using the argument number and the name of an API was only able to identify 64.7% of the deprecated Android APIs. Lastly, we found that 85.9% of the deprecated APIs can be identified by the combination of the name, the enclosing type name, the number of arguments, and the return type of the API. (See Appendix E for a list of different combinations of pieces of information from an API and how effective they are at identifying deprecated Android APIs.) It is worth mentioning that the performance of a set of heuristics in such studies cannot be mistaken for the performance of the tool using the same set of heuristics in detecting deprecated APIs from that framework or library in code examples on the web. For one thing, some deprecated APIs may appear in code examples on the web more often than the others, whereas in such studies all the deprecated APIs are treated indiscriminately. And for another, in a source code example on the web, in many cases not all information specified in the heuristics can be extracted while such studies operate in an ideal scenario. Nevertheless, the observations in such studies can still offer insights on what heuristics to use.

After the matching process, the server then returns all matches of deprecated APIs back to the visual feedback component.
5.3 Providing Visual Feedback

The extension receives the result sent back from the server. It parses the result and visually displays the result to the user accordingly.

The visual feedback can be designed and implemented in several ways. We considered the benefits and disadvantages of the following approaches. First, deprecated APIs can be highlighted inline in the code snippet in some way. It is possible to mimic IDEs like ECLIPSE to strike a line through an API to signify that that API has been deprecated. This has the benefit of creating familiarity to developers who are accustomed to working in IDEs. However, in cases when code snippets already contain strikethroughs, it becomes hard for a user to tell whether a strikethrough is done by the person who originally wrote the source code example or by the tool. The second option is to provide the feedback outside the code example. This option can 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. In short code examples, it would be rather easy for the user to spot the deprecated APIs. The last option would be combining the two options above by both providing both inline annotations directly in the code example and text descriptions outside the code example.

5.4 Deprecation Watcher

We built a prototype tool named Deprecation Watcher based on the framework described above. Deprecation Watcher detects whether code examples in posts tagged with “android” on Stack Overflow contain deprecated Android API methods. This tool is implemented as a Google Chrome extension.

Deprecation Watcher displays a message under a code example if deprecated API usages are detected. Figure 5.7 shows a code example containing deprecated APIs before activating Deprecation Watcher. Figure 5.8 shows the same code example with Deprecation Watcher turned on. A warning message has been inserted under the code example.
Had to change a bit of the tutorial on **Content Providers** since it referenced deprecated classes, this might help.

```java
import android.provider.ContactsContract.Contacts;
import android.database.Cursor;

// Form an array specifying which columns to return, you can add more.
String[] projection = new String[] {
    ContactsContract.Contacts.DISPLAY_NAME,
    ContactsContract.CommonDataKinds.Phone,
    ContactsContract.CommonDataKinds.Email
};

Uri contacts = ContactsContract.Contacts.CONTENT_LOOKUP_URI;
// id of the Contact to return.
long id = 3;

// Make the query.
Cursor managedCursor = managedQuery(contacts,
    projection, // Which columns to return
    null,      // Which rows to return (all rows)
    // Selection arguments (with a given ID)
    ContactsContract.Contacts._ID  = "id",
    // Put the results in ascending order by name
    ContactsContract.Contacts.DISPLAY_NAME + " ASC";
```

Figure 5.7: A code example on **Stack Overflow**.
Had to change a bit of the tutorial on **Content Providers** since it referenced deprecated classes, this might help.

```java
import android.provider.ContactsContract.Contacts;
import android.database.Cursor;

// Form an array specifying which columns to return, you can add more.
String[] projection = new String[] {
    ContactsContract.Contacts.DISPLAY_NAME,
    ContactsContract.CommonDataKinds.Phone
    ContactsContract.CommonDataKinds.Email
};

Uri contacts = ContactsContract.Contacts.CONTENT_LOOKUP_URI;
// id of the Contact to return.
long id = 3;

// Make the query.
Cursor managedCursor = managedQuery(contacts,
    projection, // Which columns to return
    null,       // Which rows to return (all rows)
    // Selection arguments (with a given ID)
    ContactsContract.Contacts._ID = "id",
    // Put the results in ascending order by name
    ContactsContract.Contacts.DISPLAY_NAME + " ASC";
```

**The code block above uses a deprecated API (Deprecation Watcher)**

- Cursor managedQuery(Uri, String[], String, String[], String) in class android.app.Activity was deprecated in API level 11

---

Figure 5.8: **Deprecation Watcher** identified the usage of a deprecated API in the code example and appended a warning message below it.
We chose to build a tool around Stack Overflow and the Android API for the following considerations. First, Android has been under active development and its API changes frequently. For instance, from API level 2 to API level 21 there were 569 unique deprecated API methods in total according to its official diff files. We chose Stack Overflow because it is the most popular Q&A site among developers. There are more than 690,200 questions tagged with Android as of June 2015. Many of the answers to the questions contain source code examples demonstrating how to use Android APIs to accomplish certain tasks. Second, the timeline of the first release of the Android API and the launch of Stack Overflow overlap nicely. Android API level 1 was released on September 23, 2008 and Stack Overflow was officially launched on September 15, 2008. Therefore developers were able to ask questions about Android API on Stack Overflow since the Android API’s first release. Indeed, a search of questions tagged Android on Stack Overflow shows that the first question about Android was asked on August 25, 2008 when both Android and Stack Overflow were in beta release.

The implementation of both the code element extractor and the version sensitive API matcher in Deprecation Watcher is straightforward. The code element extractor builds code element profiles from code examples in two steps. First, it uses a set of regular expressions (See Appendix F for the full set of regular expressions used in Deprecation Watcher.) to extract code elements and at the same time builds a context in the code. It then use the context to infer more information about the code elements. Concretely, it first matches each line of the code examples to the set of regular expressions to extract types, variables, and method calls. Then it infers the types of variables based on the context. For instance, in Figure 5.5, the type of dialog in line 6 is inferred to be Dialog since a variable of the same name was declared as type Dialog in line 2. We have found that the tool works better when we also include code elements from the question posts as part of the context because answer posts sometimes refer to the types and variables used in the question. As
contextInQuestion = Map()
contextInAnswer = Map()
listOfRE = List()
listOfProfiles = List()
for each code snippet in question
    reconstruct lines in code snippet
    for each line in code snippet
        extract context in line and store in contextInQuestion
    end for
end for
for each answer in answers to the question
    for each code snippet in answer
        reconstruct lines in code snippet
        for each line in code snippet
            extract context in line and store in contextInAnswer
        match listofRE to extract name, type, argument number, return type
        if type is missing
            look up type in contextInQuestion
        end if
        construct a code element profile and add to listOfProfiles
    end for
end for

Figure 5.9: Algorithm to extract code elements and construct profiles for them.

was stated in Section 5.2, the code element profile in DEPRECATION WATCHER contains the name, the enclosing type, the return type, and the number of arguments of a code element. However, the profile may only contain a subset of the above because some of the information may be absent in a code example. The code element extractor then sends a list of code element profiles to the server. An answer post may contain several source code examples. For each code example the code element extractor generate a list of code element profiles and send them to the server asynchronously.

The algorithm used in the extracting process is shown in Figure 5.9.

On the server, we first 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\(^2\). Each diff file contains the changes to Android API between two consecutive API levels. We stored information such as the name of the API, the enclosing type of the API, and the deprecated version of the API, 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. Our API matcher tries to match a profile of a code element to a deprecated API in the database based on four heuristics: the name, the argument number, the enclosing type and the return type of an API. 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 adopted a conservative strategy: we only consider there to be a match when all APIs matching the code element profile are marked as deprecated.

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 the API gets deprecated and the deprecation message for the deprecated API if there is any. Then it presents this information to users by appending this HTML segment under the code example. Figure 5.10 shows the feedback that Deprecation Watcher presents below a code example.

Figure 5.10: Feedback that Deprecation Watcher presents to the user.

\(^2\)http://javadiff.sourceforge.net/
5.5 Summary

In this chapter, we introduced our version sensitive framework to detect deprecated API usages in code examples on the web. Our framework is designed to work in the context of API evolution and it does not have assumptions about the version of an API a code example on the web is written in or the version of an API a developer uses. By using a database containing every API and its states in every version in a framework or library, our framework can be used by developers using different versions of an API.

Our framework is designed to work as an extension to a browser. It sends the information about the source code examples on a webpage to the server to match against the deprecated APIs in a target framework or library and provides visual feedback when matches are found. We also introduced our prototype tool of our framework called Deprecation Watcher. The tool is designed to detect deprecated Android API usages on Stack Overflow.
Chapter 6

Evaluation of Deprecation Watcher

In this chapter we present our evaluation of Deprecation Watcher with source code examples on the web.

By evaluating Deprecation Watcher, we want to answer the following questions:

- How well is Deprecation Watcher able to detect deprecated API usages in source code examples in terms of precision and recall?

- What factors affect how well Deprecation Watcher performs?

Section 6.1 introduces the process we used to select code snippets to evaluate. Section 6.2 provides a brief overview of the procedure we followed. Section 6.3 gives the results of our evaluation.

6.1 Selection of Code Snippets

We chose to evaluate Deprecation Watcher with posts containing code snippets on Stack Overflow. More concretely, we evaluate how well Deprecation Watcher performs on code snippets in the accepted answers to questions 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.

We chose our evaluation posts from Stack Overflow’s March 2015 data dump\(^1\) with

\(^1\)https://archive.org/details/stackexchange
the following process. We first extracted all questions having an “android” tag and their answers from the data dump. We then stored each question and its accepted answer as a single document in our database. A sample document structure is shown in Appendix G. Our database stores 638,756 documents in total. We then 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.

The above conditions are used to address the problem that the percentage of code examples containing deprecated API usages among all documents is relatively small. If we were to directly select documents randomly from all the documents, in order to obtain enough documents which actually contain deprecated API usages, we would have to use an unreasonably large number of documents to conduct our evaluation. Our filtering conditions mentioned above leverage the lexical information in a document to filter out documents that do not have deprecated API usages and that cannot possibly be detected as having deprecated API usages by the tool. They work by checking if there are any potential deprecated API usages in the code examples. To apply the first filtering condition, we obtained all the deprecated API names from our database and match them against the tokenized code examples in the accepted answer in a document. If none of the deprecated API names appear in the tokenized code examples, the code examples certainly do not contain deprecated APIs. DEPRECATION WATCHER will also not detect deprecated API usages in such cases since it first matches the names of the code elements in the code examples to names of deprecated APIs in our database. On the other hand, if deprecated API names do appear in the code examples of the accepted answer in a document, we keep track of the names because they can be used later in our evaluation. The second condition further makes use of lexical information in the whole document. It works by first obtaining the set of possible enclosing type names of the matching API names we obtained from the first filtering condition and check if any of the
type name appear in the whole document. For instance, if a document was selected because the code examples contain \texttt{getHeight}, which might be one of the two deprecated APIs with this name (\texttt{int getHeight()} in class \texttt{android.view.Display} and \texttt{int getHeight()} in class \texttt{androidrenderscript.RenderScriptGL}), then the second filtering condition ensures that at least one of the enclosing types \texttt{Display} and \texttt{RenderScriptGL} appears somewhere in the whole document. The enclosing type name of an API serves as the context for developers to understand a code example. This information may not appear in the code example itself but somewhere else in the document. To estimate how often a deprecated API is used without its enclosing type, we randomly chose 200 documents from all documents filtered with only filtering condition one. We found none of the deprecated APIs were used without their enclosing type names appearing in the document. Therefore, we can safely assume our filtering conditions will not bias our evaluation.

After applying the filtering conditions, we ended up with 7,464 documents that satisfy the above 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 chose 200 documents out of the 7,464 documents for our evaluation.

6.2 Procedure

To evaluate \textsc{Deprecation Watcher}, we first constructed the URL for each question in the 200 documents from its question ID and we went to Stack Overflow to manually examine the code snippets. We labelled them based on whether they contained deprecated \textsc{Android} API methods. We did that by manually checking the \textsc{Android} official documentation. If the API was not present in the documentation, we went on to check the API \textsc{diff} files. The absence of an API in the documentation is usually due to deletion.

This manual process is quite time consuming due to several reasons. 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 an answer can take as long as 30 minutes.

Meanwhile we also kept track of the result given by Deprecation Watcher when we loaded the webpage for the question. We examined the message given by Deprecation Watcher if there was any. We kept note of the APIs mentioned in the warning messages. Lastly, when there were indeed deprecated method APIs used in the code snippets, we kept note of whether the accepted answer or any of its comments acknowledged the usage of the deprecated APIs.

6.3 Results

6.3.1 How Well Does Deprecation Watcher Perform?

We used precision and recall to evaluate Deprecation Watcher. If an API was identified by Deprecation Watcher as deprecated and was manually classified as deprecated, it is classified as a TP (True Positive). If an API was incorrectly identified as deprecated by Deprecation Watcher, it is classified as a FP (False Positive). If an deprecated API was not correctly identified as such, it is classified as a FN (False Negative). In the evaluation we used the same conservative strategy mentioned in the last chapter for the matching process to avoid FNs: We first query the database to see whether any deprecated API satisfies the query. If so, we continue to query the database to see if all the APIs that satisfy the query are deprecated. We only consider a match to a deprecated API has been found if the above conditions are fulfilled.

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 has a precision of 1.0 and a recall of 0.86 as shown in Table 6.1.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>TP</td>
<td>FP</td>
<td>FN</td>
<td>100.0%</td>
<td>86.4%</td>
</tr>
</tbody>
</table>

Table 6.1: Deprecation Watcher has a precision of 1 and a recall of 0.86, using a conservative strategy for the API matcher.

Our result shows that Deprecation Watcher was able to detect most of the deprecated Android API usages in the code examples and that all of its reported deprecated API usages in the code examples were correct.

We also found that out of the 74 deprecated API usages Deprecation Watcher correctly detected, only 2 were acknowledged in the answers. Both of them were used deliberately after they were deprecated in order to target devices running older versions of Android. Furthermore, only 3 out of all 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 the code example before it was deprecated. A later comment to the answer pointed out this API was deprecated afterwards.

6.3.2 What Factors Affect How Well Deprecation Watcher Performs?

We have found that almost all FNs in the evaluation are due to 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 Deprecation Watcher performs in recall are as follows.

Chained Method Call. 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. Figure 6.1 is one of the source code examples that contain chained method calls. 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 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.

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

Figure 6.1: This code snippet has two chained method API calls. Deprecation is not able to get the enclosing types of the two methods inside the chained method calls.

**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 C, Deprecation Watcher would not know that method C 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 6.2, the return type of API `getLastNonConfigurationInstance` in class `android.app.Activity` has been type cast from `Object` to `int`. As a result, Deprecation Watcher misconstructed the code element profile for the code element and consequently the API matcher was not able to return a match.
int index = 0;
if (getLastNonConfigurationInstance() != null) {
    index = (Integer) getLastNonConfigurationInstance();
}

Figure 6.2: A code snippet with typecasting. DEPRECATION WATCHER fails to identify `getLastNonConfigurationInstance` as a deprecated API because its return type has been type cast to `int` from `Object`.

6.4 Evaluating Alternative Heuristics

The heuristics we used in the evaluation are the name, the argument number, the enclosing type, and the return type of a code element. Ideally, more heuristics can lead to a better result. However, as the example in Figure 6.2 shows, an extra heuristic may actually hurt the performance of the tool. In that specific example, the return type of the API `getLastNonConfigurationInstance` was type cast from `Object` to `int`. As a result, the code element constructed an incorrect profile and no match was found by the API matcher that tried to use all heuristics. This prompted us to evaluate a few other combinations of heuristics to see how they would perform on the same code examples that were used in our evaluation. More specifically, three combinations of heuristics were evaluated: the name and the argument number of a code element; the name, the argument number, and the return type of a code element; the name, the argument number, and the enclosing type of a code element. The results of the evaluation are shown in Table 6.2.

<table>
<thead>
<tr>
<th>Heuristics</th>
<th>TP</th>
<th>FP</th>
<th>FN</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, argument number</td>
<td>52</td>
<td>1</td>
<td>36</td>
<td>98.1%</td>
<td>59.1%</td>
</tr>
<tr>
<td>name, argument number, return type</td>
<td>52</td>
<td>1</td>
<td>36</td>
<td>98.1%</td>
<td>59.1%</td>
</tr>
<tr>
<td>name, argument number, enclosing type</td>
<td>77</td>
<td>0</td>
<td>11</td>
<td>100.0%</td>
<td>87.5%</td>
</tr>
</tbody>
</table>

Table 6.2: Three different combinations of heuristics were evaluated on the same posts. The third one in the table performs better than the one we used to evaluate DEPRECATION WATCHER.

We found that the heuristics containing only the name and the argument number of a code element perform as well as the heuristics which also include the return type of a code
Both of them were able to correctly identify a case that the other missed. The first set of heuristics was able to correctly detect code element `getLastNonConfigurationInstance` in Figure 6.2 while the second set of heuristics was not because the return type was type cast to a different type. 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 that the first set was not able to. The code example\(^2\) is shown in Figure 6.3. The return type `ProgramFragment` of the code element `create` in line 3 helped detect a usage of deprecated API.

```java
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 6.3: The return type helped detect the usage of a deprecated API `create` in line 3.

The first two sets of heuristics also incorrectly detected a code element as a 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 6.4. 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` wasn’t 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 as a heuristic.

We learned two lessons from evaluating these sets of heuristics. First, more heuristics do not necessarily lead to a better result. According to our evaluation, 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

\(^2\)http://stackoverflow.com/a/9353027/1062364

\(^3\)http://stackoverflow.com/a/16373967/1062364
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;
    // ... the rest of the code
    }
}

Figure 6.4: setTexture was incorrectly detected as a usage of a deprecated API in this code example because the type of page was not used.

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 implementation of the tool matters. Had the code element extractor considered the possibility that the return type of an API can be type cast to another type beforehand, it would not have misconstrued a code profile in such cases. In that case, the return type would still remain as a useless heuristic but would not hurt the performance of the tool. This has a greater implication: It can be hard to add complicated heuristics to Deprecation Watcher even though in theory they may improve the performance. The reason is that complicated heuristics tend to call for a complicated implementation that is error prone. If not well implemented, the addition of a heuristic may actually hurt the performance of the tool. This is a factor we should remind ourselves of when considering adding new heuristics into Deprecation Watcher.

6.5 Summary

In this chapter we evaluated Deprecation Watcher with Android API and code examples on Stack Overflow. We used precision and recall to evaluate the tool. Our result shows that our tool works fairly well with a precision of 1 and a recall of 0.86. Our
evaluation also informs us of situations where Deprecation Watcher does not work well so it can be improved in the future. We also evaluated three different set of heuristics on the same code examples to see how they perform. We learned two lessons from the results. First, using more heuristics does not necessarily lead to a better result. Second, we should consider the cost and benefit in adding new heuristics to improve Deprecation Watcher in the future.
Chapter 7

Discussion

In this chapter we present a discussion of several topics around our empirical study on API deprecation and our proposed framework to detect deprecated API usages online. Section 7.1 discusses a possible cause of the lack of useful information in deprecation messages and a potential fix. Section 7.2 discusses the seemingly paradoxical practice of developers of frameworks and libraries regarding removing APIs. Section 7.3 discusses why Deprecation performs fairly well. Section 7.4 discusses the usefulness of web search in software development.

7.1 Deprecation Messages

Our findings about deprecation messages echo what Robbes et al. [2012] have found in two Smalltalk software systems: nearly half of the deprecation messages do not provide information on how to replace the deprecated APIs. Our study on deprecation messages is more in depth and found that deprecation messages do even worse in offering the rationale or the expected removal date for a deprecation. This seems to be a universal problem that is more related to the habit of the developers of frameworks and libraries rather than the idiosyncrasies of different programming languages.

A possible cause for this is that there is no fixed format for deprecation messages. As a result, developers only include what they think is necessary. Therefore, it can be beneficial to establish a fixed format for deprecation messages. Considering the obstacles of forcing developers to conform to such format manually, a tool can be created to help them with the messages by asking framework or library developers several questions when they deprecate an API and the tool can generate a deprecation message automatically based on the answers.
The tool can ask the following questions:

- Is there a replacement for the deprecated API?
- If so what is the replacement?
- Why is the API getting deprecated?
- When is this API expected to be removed?

It forces developers of libraries and frameworks to put more thought into the deprecation of an API by asking a series of questions and as a result better deprecation messages can be created. There are additional benefits for enforcing a fixed deprecation message format. It enables the creation of tools that can easily analyze deprecation messages. For instance, a tool will be able to find all the deprecated APIs with an replacement and replace them automatically or generate a report for all the deprecations in a system and prioritize them based on information in the deprecation messages.

We also believe it would not create much more overhead for developers of frameworks and libraries for the following reasons. First, API deprecation does not happen in every release of a system and when it does happen in a release, in most cases, not a lot of APIs get deprecated. Furthermore, we observed in the systems we studied, many APIs get deprecated due to the same change and can be removed in the future in the same version. If implemented properly, a tool can use this information and reduce the workload of the developers.

7.2 Removing APIs

One of our findings is that most of the systems in our study are reluctant to remove deprecated APIs in their systems. Yet at the same time we found that a lot of APIs got removed without being deprecated first.
7.2.1 Removing APIs Without Deprecating Them First

Removing APIs without deprecating them first may seem dangerous because it can break a client’s code without warning. Two factors may have contributed to this phenomenon. First, it is often unrealistic to deprecate APIs and keep them around 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 of the systems when breaking changes occurred, we found 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,\textsuperscript{1} in the removed classes section, the developers state “removed unused class AbstractUntypedCollectionDecorator”. And similarly in the release notes of COMMONS LANG 3.2.1,\textsuperscript{2}, the developers state “it is assumed that this change will not break clients” when they removed three protected methods.

It is without doubt that developers of the systems have much knowledge about how their systems are used, or at least, should be used. However, they can also make false assumptions because their knowledge is limited. As the phenomenon of remove–resurrect–deprecate in our study demonstrates, they may remove APIs that are actually used and have to add them back.

We speculate that it is common for the developers of frameworks and libraries to retire APIs that they assume are used by users by deprecating them first and remove APIs that they think are not used directly without deprecating them first. We manually examined several release notes of some versions of the systems where APIs got removed without being deprecated first and found that they fit our speculation. However, so far our work is still incomplete and preliminary; therefore, the validity of this hypothesis still needs further investigation in the future.

\textsuperscript{1}\url{http://commons.apache.org/proper/commons-collections/release_4_0.html}
\textsuperscript{2}\url{http://commons.apache.org/proper/commons-lang/release-notes/RELEASE-NOTES-3.2.1.txt}
7.2.2 Removing Deprecated APIs

Our result shows 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. However, different systems have different conventions about major releases. 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. For instance, in Struts, version 1.2.0 is considered a major release.

We argue that adopting semantic versioning [sem] is good for both developers and users of the systems. Semantic versioning is a version scheme in which starting from version 1.0.0, any backward incompatible changes such as APIs being removed should always increment the major version number. In this way, developers would easily estimate whether it is safe to upgrade the frameworks and libraries they use based on changes in the release numbers. This in turn makes it easy for developers of frameworks and libraries to make plans to remove deprecated APIs. In our study, many systems already conform to the practice of semantic versioning regrading removing deprecated APIs. JUnit has started to adopt the principles of semantic versioning as its deprecation policy\(^3\) as well.

7.3 The Performance of DEPRECATION WATCHER

In our evaluation, Deprecation Watcher works fairly well despite the fact that it only extracts limited information of the code elements in a code example. This can be attributed to the fact that code examples on Stack Overflow are intentionally kept simple and self-contained so it can be easily understood. As a result, even with a straightforward implementation, Deprecation Watcher is able to perform well. On the other hand, Deprecation Watcher is unlikely fare as well in some other situations. For instance, if a code example is not quite self-contained and more information need to be inferred, the simple inference rule

\(^3\)https://github.com/junit-team/junit/wiki/Deprecation-Policy
used in Deprecation Watcher may not be sufficient. Furthermore, if a code example is very complicated, the simple regular expression matching rules in Deprecation Watcher may not suffice as well.

7.4 The Usefulness of Web Search in Software Development

Web search has proved to be valuable to developers during software development tasks. It helps them find helpful resources on sites like Stack Overflow and programming blogs.

However, as is suggested by our study, web search also has its problems, caused by two closely related factors: code examples on the web can be written with different versions of an API, and developers can be using different versions of the API as well. When APIs get 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 takes 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 proposed 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.

7.5 Summary

In this chapter, we discussed several topics that are closely related to our work in this thesis. In the next chapter, we will examine the threats to the validity of our work.
Chapter 8

Threats to Validity

In this chapter we discuss the threats to the validity of our study. We first discuss the threats to the validity of our data analysis on API deprecation in Section 8.1. Then in Section 8.2 we discuss the threats to the validity of our evaluation of Deprecation Watcher.

8.1 Threats to the Validity of the API Deprecation Study

The accuracy of our data analysis on API deprecation depends on the following factors: the choices of third party libraries, the versions of the libraries used in our analysis, and the accuracy of the tools we used in our analysis.

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. And lastly, we did not include a few libraries because our tools were not able to work on them.

The versions of the libraries used in our analysis. We tried to collect as many versions of the source code and binary code of the libraries as possible in our study so that we can pinpoint when APIs got deprecated and when they got removed or un-deprecated. But 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 make sure the difference between the reported version and the version an API deprecation or removal actually occurred is small in these cases.

**The accuracy of the tools used in our analysis.** In our analysis, we used two tools and they play an important role in our analysis. We used **QDox** to get the set of deprecated APIs in each version of the libraries through parsing their source code of that version. We used **Clirr** to verify if an API got removed. The accuracy of our analysis is affected by how well the tools we used work. **QDox** has a larger effect on our analysis since **Clirr** is used primarily for verifying. During our analysis, we randomly selected some of the results given by the tools and double-checked them by manually looking at relevant source code files. We found the output of the tools to be accurate. However, since we could not manually check all their output, the accuracy of the tools still remains as a threat to the validity of our analysis.

### 8.2 Threats to Validity of the Evaluation of Deprecation Watcher

The accuracy of our evaluation of Deprecation Watcher is mainly affected the choice of code examples for evaluation.

**The choice of posts with which we evaluate Deprecation Watcher.** We selected posts that possibly contain deprecated APIs through two criteria mentioned in Chapter 5. 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.

8.3 Summary

In this chapter we discussed the threats to the validity of our study. Our data analysis on API deprecation is affected by the frameworks and libraries we use, the versions of the frameworks and libraries and the tools we used to get API deprecations and breaking API changes. The threat to the validity of our evaluation of the tool Deprecation Watcher lies in our choices of posts. Our filtering condition may filter out posts containing deprecated API usages in rare cases.
Chapter 9

Conclusions and Future Work

9.1 Conclusions

Our work has two goals. First, we set out to answer several important questions on API deprecation and second, we aim to solve the problem that developers can waste significant amount of time on code examples containing deprecated API usages on the web.

In the first part of this thesis, we used the history data of API deprecation and removal in 26 open source Java third party frameworks and libraries to examine how deprecation is being used. We found that API deprecation is used by all the systems. We also found that API deprecation is underused, because many APIs were removed without being deprecated first. Our classification of deprecation messages according to three criteria (whether they refer to a concrete replacement, whether they offer an explanation for the deprecation and whether they specify a timeline for the removal of the deprecated API) shows that deprecation messages do not excel at any of them. Just over half of them offered concrete replacements for their deprecated APIs. Only 9.1% of the deprecation messages provided a rationale for the deprecated API and merely 5.7% of them specified a concrete time frame for the removal of the deprecated API.

We found that un-deprecation happened in almost half of the systems we studied and an interesting phenomenon of removal–resurrection–deprecation of APIs occurred in three systems as well. Surprisingly, more than 40% of all deprecated APIs got removed even though 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 & removed APIs were removed either before version 1.0 or during the transition to a major release and few APIs were removed during other versions.
Our results have several implications. First, developers of frameworks and libraries need to provide better deprecation messages. Second, they need tool support to get a better sense of how their APIs are used so they can have more confidence removing deprecated APIs from their systems. Third, the reluctance to removing deprecated APIs, shown by more than half of the systems, and the practice of removing deprecated APIs almost exclusively before version 1.0 and during the transition to a major release show that developers of frameworks and libraries take backward compatibility of their systems seriously. Fourth, users of frameworks and libraries should take caution to update to a major release of a framework or library.

In the second part of this thesis, 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 aware. As a result, our framework is well suited in the context of API evolution. By making use of the API change history, we could have also made our framework more useful by detecting removed APIs as well. We chose not to do so because our research specifically focuses on API deprecation. 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 was able to achieve 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 work has the following two main contributions:

• We conducted an empirical study on the practice of API deprecation in 26 open source Java third party frameworks and libraries. As far as we know, no study that focuses on API deprecation of this scale has been conducted. We answered our five research questions regrading how API deprecation has been used and how APIs continue to change after
being deprecated. Our answers can benefit API developers, API users as well as other researchers in this field.

- We proposed a version sensitive framework to detect deprecated API usages in source code examples on the web. We also implemented the framework as a prototype tool called Deprecation Watcher to specifically detect deprecated Android API usages in code examples on Stack Overflow. Our evaluation demonstrated that the tool works well with a straightforward implementation.

9.2 Future Work

9.2.1 Investigate Why Systems Remove APIs Without Deprecating Them First

We found that developers of the systems in our study are obviously concerned with backward incompatibility yet many APIs were removed without being deprecated first, possibly resulting in broken client code. We think this paradox is worth further investigation.

We manually checked a few relevant release notes and found that developers 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 by users and the removal of these APIs would not affect users of the systems. In several systems we also found that most of the removed APIs in question were protected instead of public. Developers of the systems may believe that no user would inherit the classes in question and went forward to remove protected APIs. Therefore, we speculate that developers of frameworks and libraries remove APIs without deprecating first when they think these APIs are not used by users. Our speculation needs further study.

9.2.2 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 class hierarchy. As a result, it is not able to extract some important information from a code example. There are two ways we could improve this. 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 approach may complicate the implementation of the tool significantly. Alternatively, we can choose to adopt a parser to extract information from a code example if it can lead to a better performance. These approaches can potentially improve the performance of the tool in recall.

Additionally, in our evaluation of Deprecation Watcher we discovered a few cases where the names of variables in a code example can be used to infer their types. Developers tend to use abbreviations or a term very close to a type as the variable name of that type. For example, we observed that developers tend to name a variable of type TextToSpeech “tts”, and a variable of type Drawable “drawable”. We can devise a set of rules to map a variable to a type when the declaration of that variable cannot be found. Then the types can be used to help the API matching process.

9.2.3 Evaluate Deprecation Watcher with Developers

We evaluated Deprecation Watcher in terms of precision and recall but with the involvement of developers, we can further evaluate the usability of the tool. For example, Deprecation Watcher aims to give timely feedback to developers as they view a webpage since the efficiency of Deprecation Watcher may affect how useful it is perceived. Furthermore, we can also investigate what information is most useful to developers when a deprecated API is detected so the feedback component of Deprecation Watcher can be improved as well.
Appendix A

Systems Used in Data Analysis

Table A.1 lists the JAVA third party frameworks and libraries we used in our study. The versions in the column called first version are the first versions used in our study, which may not be the actual first versions of the systems. The reason is that we were not able to locate the binary and source code of the first versions of some systems despite our best effort. The versions in the column called last version are the last versions of the systems used in our study. They were the latest versions of the systems as the time of our study. The number of the years between the release of a system’s first version used in our study and its last version used in our study is recorded in the last column of the table.
<table>
<thead>
<tr>
<th>System</th>
<th>First version</th>
<th>Last version</th>
<th>Classes</th>
<th>Versions</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>APACHE-ANT</td>
<td>1.1</td>
<td>1.9.4</td>
<td>93 to 1127</td>
<td>29</td>
<td>14</td>
</tr>
<tr>
<td>APACHE-DERBY</td>
<td>10.1.1.0</td>
<td>10.11.1.1</td>
<td>1758 to 3145</td>
<td>23</td>
<td>9</td>
</tr>
<tr>
<td>COMMONS-BEANUTILS</td>
<td>1.5</td>
<td>1.9.2</td>
<td>63 to 119</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>COMMONS-CLI</td>
<td>1.0</td>
<td>1.2</td>
<td>19 to 21</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>COMMONS-CODEC</td>
<td>1.1</td>
<td>1.10</td>
<td>14 to 76</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>COMMONS-COLLECTIONS</td>
<td>1.0</td>
<td>4.0</td>
<td>36 to 411</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>COMMONS-CONFIGURATION</td>
<td>1.1</td>
<td>2.0-alpha2</td>
<td>66 to 296</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>COMMONS-FILEUPLOAD</td>
<td>1.0-beta-1</td>
<td>1.3.1</td>
<td>7 to 47</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>COMMONS-IO</td>
<td>1.0</td>
<td>2.4</td>
<td>30 to 108</td>
<td>13</td>
<td>8</td>
</tr>
<tr>
<td>COMMONS-LANG</td>
<td>1.0</td>
<td>3.3.2</td>
<td>28 to 179</td>
<td>18</td>
<td>13</td>
</tr>
<tr>
<td>COMMONS-LOGGING</td>
<td>1.0</td>
<td>1.2</td>
<td>11 to 17</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>GSON</td>
<td>1.1</td>
<td>2.3.1</td>
<td>104 to 84</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>H2</td>
<td>1.2.122</td>
<td>1.4.185</td>
<td>449 to 572</td>
<td>64</td>
<td>6</td>
</tr>
<tr>
<td>JDOM</td>
<td>b3</td>
<td>2.0.5</td>
<td>29 to 164</td>
<td>19</td>
<td>13</td>
</tr>
<tr>
<td>JODA-TIME</td>
<td>1.5</td>
<td>2.7</td>
<td>221 to 241</td>
<td>14</td>
<td>8</td>
</tr>
<tr>
<td>JUNIT</td>
<td>3.8.1</td>
<td>4.12</td>
<td>56 to 206</td>
<td>22</td>
<td>12</td>
</tr>
<tr>
<td>LOG4J</td>
<td>1.0.4</td>
<td>2.0.2</td>
<td>167 to 70</td>
<td>30</td>
<td>13</td>
</tr>
<tr>
<td>LOGBACK</td>
<td>0.2</td>
<td>1.1.1</td>
<td>169 to 563</td>
<td>57</td>
<td>8</td>
</tr>
<tr>
<td>MAVEN-MODEL</td>
<td>2.0-alpha-3</td>
<td>3.2.5</td>
<td>38 to 50</td>
<td>43</td>
<td>9</td>
</tr>
<tr>
<td>MAVEN-PLUGIN-API</td>
<td>2.0-alpha-3</td>
<td>3.2.5</td>
<td>5 to 24</td>
<td>41</td>
<td>9</td>
</tr>
<tr>
<td>MOCKITO-ALL</td>
<td>1.8.0-rc2</td>
<td>2.0.2-beta</td>
<td>429 to 613</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>NEKOHTML</td>
<td>0.1</td>
<td>1.9.21</td>
<td>9 to 47</td>
<td>57</td>
<td>12</td>
</tr>
<tr>
<td>PLEXUS-UTILS</td>
<td>1.0.4</td>
<td>3.0.21</td>
<td>61 to 96</td>
<td>59</td>
<td>9</td>
</tr>
<tr>
<td>SLF4J</td>
<td>1.1.0-RC0</td>
<td>1.7.7</td>
<td>11 to 27</td>
<td>40</td>
<td>8</td>
</tr>
<tr>
<td>STRUTS</td>
<td>1.2.2</td>
<td>2.3.16.2</td>
<td>286 to 404</td>
<td>48</td>
<td>12</td>
</tr>
<tr>
<td>XERCESIMPL</td>
<td>2.4.0</td>
<td>2.11.0</td>
<td>510 to 706</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Table A.1: Systems used in study.
Appendix B

Underuse of API Deprecation

We found that the number of removed API is far greater than the number of deprecated APIs in 18 systems. Removing APIs before deprecating them indicates API deprecation has been underused.

<table>
<thead>
<tr>
<th>Library</th>
<th># removed APIs</th>
<th># overlapping APIs</th>
<th># deprecated APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>APACHE-ANT</td>
<td>198</td>
<td>1</td>
<td>110</td>
</tr>
<tr>
<td>APACHE-DERBY</td>
<td>1812</td>
<td>0</td>
<td>71</td>
</tr>
<tr>
<td>COMMONS-BEANUTILS</td>
<td>2</td>
<td>0</td>
<td>46</td>
</tr>
<tr>
<td>COMMONS-CLI</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>COMMONS-CODEC</td>
<td>1</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>COMMONS-Collections</td>
<td>60</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>COMMONS-CONFIGURATION</td>
<td>35</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>COMMONS-FILEUPLOAD</td>
<td>23</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>COMMONS-IO</td>
<td>1</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>COMMONS-LOGGING</td>
<td>2</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>GSON</td>
<td>24</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>JODA-TIME</td>
<td>18</td>
<td>18</td>
<td>32</td>
</tr>
<tr>
<td>JUNIT</td>
<td>103</td>
<td>0</td>
<td>26</td>
</tr>
<tr>
<td>MAVEN-MODEL</td>
<td>140</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>MAVEN-PLUGIN-API</td>
<td>20</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>MOCKITO-ALL</td>
<td>176</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>NEKOHTML</td>
<td>32</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PLEXUS-UTILS</td>
<td>22</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>SLF4J</td>
<td>8</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>XERCESImpl</td>
<td>405</td>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>STRUTS</td>
<td>130</td>
<td>65</td>
<td>79</td>
</tr>
<tr>
<td>JDOM</td>
<td>83</td>
<td>83</td>
<td>96</td>
</tr>
<tr>
<td>COMMONS-LANG</td>
<td>111</td>
<td>29</td>
<td>43</td>
</tr>
<tr>
<td>LOG4J</td>
<td>172</td>
<td>85</td>
<td>87</td>
</tr>
<tr>
<td>LOGBACK</td>
<td>286</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

Table B.1: The number of APIs that have been removed, deprecated, and both deprecated and removed.
Appendix C

Analysis of Deprecation Messages

The following table classifies the deprecation messages of each library with the criteria mentioned above. The library LOGBACK-ACCESS is excluded from the table since it doesn’t have any deprecated APIs.
<table>
<thead>
<tr>
<th>Library</th>
<th>Replacement</th>
<th>Explanation</th>
<th>Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N</td>
<td>F</td>
<td>V</td>
</tr>
<tr>
<td>APACHE-ANT</td>
<td>1</td>
<td>45</td>
<td>14</td>
</tr>
<tr>
<td>APACHE-DERBY</td>
<td>1</td>
<td>69</td>
<td>1</td>
</tr>
<tr>
<td>COMMONS-BEANUTILS</td>
<td>0</td>
<td>8</td>
<td>35</td>
</tr>
<tr>
<td>COMMONS-CLI</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>COMMONS-CODEC</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>COMMONS-COLLECTIONS</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>COMMONS-CONFIGURATION</td>
<td>0</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>COMMONS-FILEUPLOAD</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>COMMONS-IO</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>COMMONS-LOGGING</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>GSON</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>H2</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>JODA-TIME</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>JUNIT</td>
<td>2</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>MAVEN-MODEL</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>MAVEN-PLUGIN-API</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MOCKITO-ALL</td>
<td>0</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>NEKOHTML</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PLEXUS-UTILS</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SLF4J</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>XERCESIMPL</td>
<td>6</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>STRUTS</td>
<td>1</td>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td>JDOM</td>
<td>0</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>COMMONS-LANG</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>LOG4J</td>
<td>0</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>LOGBACK</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Table C.2: Classification of deprecated messages based on three different criteria: whether the message offers a replacement (N: without replacement, F: no mention of a replacement, V: a vague replacement, C: a concrete replacement), whether the message offers an explanation (F: no explanation, T: with an explanation) and whether the message has a removal plan. (F: no plan, V: a vague plan, C: a concrete plan)
Appendix D

Remove–Resurrect–Deprecate

Compared to the *deprecate–replace–remove* cycle to use deprecation to evolve APIs, we found the usage of *remove–resurrect–deprecate* to resurrect and deprecate removed APIs in 3 systems we studied. This interesting phenomenon indicates developers lack the full knowledge of how their APIs are used by their users. Also note that there could be cases where a removed API is simply resurrected but not deprecated and we didn’t investigate such cases in our research.

Table D.1 shows such cases in the systems.

<table>
<thead>
<tr>
<th>System</th>
<th>Class</th>
<th>Signature</th>
<th>Removed version</th>
<th>Resurr. version</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUNIT</td>
<td>org.junit.runners.BlockJUnit4ClassRunner</td>
<td>protected java.util.List rules (java.lang.Object)</td>
<td>4.8.2</td>
<td>4.9</td>
</tr>
<tr>
<td>JUNIT</td>
<td>org.junit.internal.RealSystem</td>
<td>public void exit (int)</td>
<td>4.11-beta-1</td>
<td>4.12-beta-2</td>
</tr>
<tr>
<td>LOG4J</td>
<td>org.apache.log4j.helpers.Loader</td>
<td>public java.net.URL getResource(java.lang.String, java.lang.Class)</td>
<td>1.2.1</td>
<td>1.2.12</td>
</tr>
<tr>
<td>XERCESIMPL</td>
<td>org.apache.xerces.dom.CoreDocumentImpl</td>
<td>public void setVersion (java.lang.String)</td>
<td>2.6.1</td>
<td>2.8.0</td>
</tr>
<tr>
<td>XERCESIMPL</td>
<td>org.apache.xerces.dom.CoreDocumentImpl</td>
<td>public void setEncoding (java.lang.String)</td>
<td>2.6.1</td>
<td>2.8.0</td>
</tr>
<tr>
<td>XERCESIMPL</td>
<td>org.apache.xerces.dom.CoreDocumentImpl</td>
<td>public boolean getStandalone()</td>
<td>2.6.1</td>
<td>2.8.0</td>
</tr>
<tr>
<td>XERCESIMPL</td>
<td>org.apache.xerces.dom.CoreDocumentImpl</td>
<td>public java.lang.String getVersion()</td>
<td>2.6.1</td>
<td>2.8.0</td>
</tr>
<tr>
<td>XERCESIMPL</td>
<td>org.apache.xerces.dom.CoreDocumentImpl</td>
<td>public java.lang.String getEncoding()</td>
<td>2.6.1</td>
<td>2.8.0</td>
</tr>
<tr>
<td>XERCESIMPL</td>
<td>org.apache.xerces.dom.CoreDocumentImpl</td>
<td>public void setStandalone (boolean)</td>
<td>2.6.1</td>
<td>2.8.0</td>
</tr>
</tbody>
</table>

Table D.1: Three systems resurrected and deprecated some of their APIs after removing them.
Appendix E

Different Combinations of Pieces of Information of an API in Identifying Deprecated Android APIs

In order to see what heuristics may be helpful, we used a few pieces of information of an API to see how well they can identifying the deprecated Android APIs from all Android APIs. Table E.1 shows the pieces of information used and the percentage of deprecated Android APIs they are able to uniquely identify, and on average how many Android APIs can match to a deprecated Android API based on the information. Not surprisingly, our result shows that using more pieces of information helps uniquely identify deprecated APIs. We also find that certain piece of information helps more. For example, using the enclosing type of an deprecated Android API is more effective than using the return type or the argument number of an API.

<table>
<thead>
<tr>
<th>Pieces of information used</th>
<th>Percentage</th>
<th># Matching APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>0.53</td>
<td>4.44</td>
</tr>
<tr>
<td>name, return type</td>
<td>0.59</td>
<td>2.66</td>
</tr>
<tr>
<td>name, argument number</td>
<td>0.65</td>
<td>2.58</td>
</tr>
<tr>
<td>name, enclosing type</td>
<td>0.78</td>
<td>1.34</td>
</tr>
<tr>
<td>name, return type, argument number</td>
<td>0.68</td>
<td>2.11</td>
</tr>
<tr>
<td>name, enclosing type, return type</td>
<td>0.80</td>
<td>1.31</td>
</tr>
<tr>
<td>name, enclosing type, argument number</td>
<td>0.85</td>
<td>1.17</td>
</tr>
<tr>
<td>name, enclosing type, argument number, return type</td>
<td>0.86</td>
<td>1.15</td>
</tr>
</tbody>
</table>

Table E.1: More deprecated Android APIs can be uniquely identified by using more pieces of information of an API but the usefulness of different pieces of information differ.
Appendix F

Regular Expressions Used in Deprecation Watcher

Deprecation Watcher uses a set of regular expressions to match each line of the code snippet to extract types, variables, constructor invocations and method calls.

The regular expression in Figure F.1 matches variable declarations and definitions. The variable and type pairs will be stored in the context of the code snippet. When the variable is used later, its type can be obtained from the context.

```plaintext
1  // This regex deals with class types with . in them like ProgramFragment.Builder
2  var typeVariableRegex = /\w+(?:\:\:\\w+)?\s(b[a-z]\w{0,})/g;
```

Figure F.1: A regular expression that matches variable declarations and definitions.

The regular expression in Figure F.2 matches constructor and method names.

```plaintext
1  var methodRegex = /\(b[a-z]\w+\)/g;
```

Figure F.2: A regular expression that matches method and constructor names.

The regular expression in Figure F.3 matches a method call.

```plaintext
1  var callerMethodRegex = /\(\w+\)\.\(\w+\)\(/g;
```

Figure F.3: A regular expression that matches method calls.

Besides those regular expressions, Deprecation Watcher also tries to determine the number of arguments via a regular expression whenever it is possible.
Appendix G

Document Structure in Database

Here, we illustrate how we store a question and its accepted answer from Stack Overflow in our database.

```
1  {
2    "id" : ObjectId("558ce567d1aaee25c17f57d7"),
3    "body" : "<p>I need to know how to run an Android emulator automatically from Eclipse.<br>Any ideas?</p>",
4    "viewCount" : 57807,
5    "title" : "How do I run an Android emulator automatically from Eclipse?",
6    "lastEditorUserId" : "321731",
7    "lastActivityDate" : ISODate("2014-02-04T20:26:45.053Z"),
8    "lastEditDate" : ISODate("2014-02-04T20:26:45.053Z"),
9    "answers" : [
10       {
11           "body" : "<p>Once you have installed <a href="http://code.google.com/android/intro/installing.html#installing_plugin">ADT</a>, you need to define an <a href="http://code.google.com/android/intro/hello-android.html#run">ADT launch configuration</a>, in order for your project to execute itself as an Android executable.<br><br></p>
12              <p><img src="http://i.stack.imgur.com/8Fdv6.png" alt="Hello World Android"></p>
13              <p>Then, do not forget to <code>Go to Eclipse-&gt;Preferences-&gt;Run/Debug-&gt;Launching</code>.<br>In the <code>Launch
```
Operation</code> subsection, select <code>Always launch the previously launched application</code>:</p>

Eclipse launching options”>

```

"lastEditorDisplayName" : "VonC",
"lastEditorUserId" : "6309",
"postId" : "2",
"lastEditDate" : ISODate("2012-08-29T05:32:17.320Z"),
"lastActivityDate" : ISODate("2012-08-29T05:32:17.320Z"),
"commentCount" : "5",
"score" : 39,
"ownerDisplayName" : "VonC",
"parentId" : "506777",
"ownerUserId" : "6309",
"creationDate" : ISODate("2009-02-03T11:44:38.630Z"),
"id" : "506801"
}
```

```

"answerCount" : 1,
"score" : 20,
"ownerDisplayName" : "tantan",
"postId" : "1",
"commentCount" : "0",
"creationDate" : ISODate("2009-02-03T11:35:25.730Z"),
"favoriteCount" : 5,
"id" : "506777",
"tags" : [
"android",
```
"eclipse",
"emulator"
]}
}
Appendix H

Evaluation Data Set

This chapter includes the 200 Stack Overflow posts used in evaluation. These questions were randomly chosen from 7,464 posts.

Table H.1: Questions used in evaluation. They were randomly chosen from 7464 posts.

<table>
<thead>
<tr>
<th>Question ID</th>
<th>Candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>14123580</td>
<td>getIntent</td>
</tr>
<tr>
<td>14272295</td>
<td>setLockingEnabled</td>
</tr>
<tr>
<td>22296415</td>
<td>setTicker</td>
</tr>
<tr>
<td>11444622</td>
<td>removeGlobalOnLayoutListener</td>
</tr>
<tr>
<td>7149366</td>
<td>removeGlobalOnLayoutListener</td>
</tr>
<tr>
<td>10055667</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>9731914</td>
<td>getDefault, sendTextMessage</td>
</tr>
<tr>
<td>9864684</td>
<td>getIntent</td>
</tr>
<tr>
<td>22772632</td>
<td>removeGlobalOnLayoutListener</td>
</tr>
<tr>
<td>11819229</td>
<td>speak</td>
</tr>
<tr>
<td>6252029</td>
<td>create</td>
</tr>
<tr>
<td>7945346</td>
<td>getLastNonConfigurationInstance</td>
</tr>
<tr>
<td>17122230</td>
<td>getIntent</td>
</tr>
<tr>
<td>12070005</td>
<td>create</td>
</tr>
<tr>
<td>22204869</td>
<td>getIntent</td>
</tr>
<tr>
<td>13667088</td>
<td>addPreferencesFromResource</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Question ID</th>
<th>Candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>7321177</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>16373550</td>
<td>onRetainNonConfigurationInstance,</td>
</tr>
<tr>
<td></td>
<td>getLastNonConfigurationInstance</td>
</tr>
<tr>
<td>10991159</td>
<td>dismissDialog, showDialog</td>
</tr>
<tr>
<td>3499910</td>
<td>getIntent</td>
</tr>
<tr>
<td>15927726</td>
<td>registerListener</td>
</tr>
<tr>
<td>19633318</td>
<td>removeGlobalOnLayoutListener</td>
</tr>
<tr>
<td>10790129</td>
<td>getIntent</td>
</tr>
<tr>
<td>22248762</td>
<td>create</td>
</tr>
<tr>
<td>4942398</td>
<td>setLatestEventInfo</td>
</tr>
<tr>
<td>14468547</td>
<td>getWidth, getHeight</td>
</tr>
<tr>
<td>12883253</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>25225859</td>
<td>getWidth, getHeight</td>
</tr>
<tr>
<td>20627672</td>
<td>onCreateDialog, showDialog</td>
</tr>
<tr>
<td>10713730</td>
<td>getRunningTasks</td>
</tr>
<tr>
<td>3592717</td>
<td>onCreateDialog, create</td>
</tr>
<tr>
<td>24430656</td>
<td>getIntent</td>
</tr>
<tr>
<td>21139674</td>
<td>getDefault, sendTextMessage</td>
</tr>
<tr>
<td>19829943</td>
<td>getBlockSize, getAvailableBlocks, setPreviewFrameRate, setCamera, setType</td>
</tr>
<tr>
<td>18362107</td>
<td>create</td>
</tr>
<tr>
<td>28664189</td>
<td>getIntent</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Question ID</th>
<th>Candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>25913077</td>
<td>create</td>
</tr>
<tr>
<td>15231987</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>27991697</td>
<td>getIntent</td>
</tr>
<tr>
<td>25108540</td>
<td>addTab, newTab, setNavigationMode</td>
</tr>
<tr>
<td>14357787</td>
<td>setLayoutAlgorithm</td>
</tr>
<tr>
<td>15402470</td>
<td>getIntent</td>
</tr>
<tr>
<td>4887267</td>
<td>removeAllCookie</td>
</tr>
<tr>
<td>23853552</td>
<td>create</td>
</tr>
<tr>
<td>7411195</td>
<td>getWidth, getHeight</td>
</tr>
<tr>
<td>12783476</td>
<td>getDefault, sendTextMessage</td>
</tr>
<tr>
<td>7189948</td>
<td>showDialog</td>
</tr>
<tr>
<td>26708828</td>
<td>speak</td>
</tr>
<tr>
<td>3757229</td>
<td>createFromPdu, sendDataMessage, getOriginatingAddress, getUserData, getDefault</td>
</tr>
<tr>
<td>17834948</td>
<td>findPreference</td>
</tr>
<tr>
<td>13169260</td>
<td>newKeyguardLock</td>
</tr>
<tr>
<td>22084492</td>
<td>getIntent</td>
</tr>
<tr>
<td>15829738</td>
<td>setRemoteAdapter</td>
</tr>
<tr>
<td>20594833</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>21533408</td>
<td>getPreferenceManager</td>
</tr>
<tr>
<td>28614168</td>
<td>getIntent</td>
</tr>
<tr>
<td>1540272</td>
<td>drawBitmap</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Question ID</th>
<th>Candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>3047453</td>
<td>getIntent</td>
</tr>
<tr>
<td>11919587</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>4039409</td>
<td>getIntent</td>
</tr>
<tr>
<td>6143056</td>
<td>getIntent</td>
</tr>
<tr>
<td>4921811</td>
<td>setBackgroundDrawable</td>
</tr>
<tr>
<td>10904933</td>
<td>create</td>
</tr>
<tr>
<td>18833688</td>
<td>setCamera</td>
</tr>
<tr>
<td>7540798</td>
<td>managedQuery</td>
</tr>
<tr>
<td>15160598</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>23577827</td>
<td>getIntent</td>
</tr>
<tr>
<td>6765465</td>
<td>setLatestEventInfo</td>
</tr>
<tr>
<td>13723236</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>22209046</td>
<td>create</td>
</tr>
<tr>
<td>26190801</td>
<td>create</td>
</tr>
<tr>
<td>26710519</td>
<td>setBackgroundDrawable</td>
</tr>
<tr>
<td>14638438</td>
<td>getIntent</td>
</tr>
<tr>
<td>22732087</td>
<td>getIntent</td>
</tr>
<tr>
<td>22011862</td>
<td>getOrientation</td>
</tr>
<tr>
<td>12543290</td>
<td>create</td>
</tr>
<tr>
<td>9720719</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>3331805</td>
<td>getMatrix</td>
</tr>
<tr>
<td>5305508</td>
<td>getIntent</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Question ID</th>
<th>Candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>4938822</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>28813746</td>
<td>create</td>
</tr>
<tr>
<td>17643167</td>
<td>addPreferencesFromResource</td>
</tr>
<tr>
<td>14067339</td>
<td>getIntent, getInputBuffers</td>
</tr>
<tr>
<td>16562335</td>
<td>getIntent</td>
</tr>
<tr>
<td>9834499</td>
<td>setCamera</td>
</tr>
<tr>
<td>8831642</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>14411477</td>
<td>getIntent</td>
</tr>
<tr>
<td>22476683</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>25505658</td>
<td>onStart</td>
</tr>
<tr>
<td>5316651</td>
<td>requery</td>
</tr>
<tr>
<td>13428356</td>
<td>getIntent</td>
</tr>
<tr>
<td>23195241</td>
<td>create</td>
</tr>
<tr>
<td>8470897</td>
<td>create</td>
</tr>
<tr>
<td>26104660</td>
<td>create</td>
</tr>
<tr>
<td>14388675</td>
<td>getIntent</td>
</tr>
<tr>
<td>13679568</td>
<td>registerListener</td>
</tr>
<tr>
<td>25509802</td>
<td>create, setTicker</td>
</tr>
<tr>
<td>16880451</td>
<td>getSelectedTab</td>
</tr>
<tr>
<td>4257265</td>
<td>registerListener</td>
</tr>
<tr>
<td>5629297</td>
<td>getIntent</td>
</tr>
<tr>
<td>7820847</td>
<td>getIntent</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Question ID</th>
<th>Candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>18160508</td>
<td>getIntent</td>
</tr>
<tr>
<td>27592627</td>
<td>onStart</td>
</tr>
<tr>
<td>9254874</td>
<td>create, setTexture</td>
</tr>
<tr>
<td>14766756</td>
<td>getRunningTasks</td>
</tr>
<tr>
<td>5786266</td>
<td>.createFromPdu, getDisplayOriginatingAddress</td>
</tr>
<tr>
<td>11608549</td>
<td>getRunningTasks</td>
</tr>
<tr>
<td>11118749</td>
<td>onCreateDialog, create, showDialog</td>
</tr>
<tr>
<td>3600326</td>
<td>createFromPdu, getDefault, sendTextMessage,</td>
</tr>
<tr>
<td></td>
<td>getOriginatingAddress, getMessageBody</td>
</tr>
<tr>
<td>21579358</td>
<td>getIntent</td>
</tr>
<tr>
<td>23867781</td>
<td>create</td>
</tr>
<tr>
<td>12692636</td>
<td>setLatestEventInfo</td>
</tr>
<tr>
<td>21915287</td>
<td>setTicker</td>
</tr>
<tr>
<td>26706063</td>
<td>getIntent</td>
</tr>
<tr>
<td>25244930</td>
<td>getIntent</td>
</tr>
<tr>
<td>20017840</td>
<td>setNavigationMode, setListAdapter</td>
</tr>
<tr>
<td>12224860</td>
<td>setDatabasePath, onExceededException</td>
</tr>
<tr>
<td>23649161</td>
<td>getIntent</td>
</tr>
<tr>
<td>13063816</td>
<td>create, setButton2</td>
</tr>
<tr>
<td>28559281</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>26404760</td>
<td>create</td>
</tr>
<tr>
<td>11582206</td>
<td>getRunningTasks</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Question ID</th>
<th>Candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>25751712</td>
<td>getIntent</td>
</tr>
<tr>
<td>7035990</td>
<td>getIntent</td>
</tr>
<tr>
<td>11925039</td>
<td>unregisterListener, registerListener</td>
</tr>
<tr>
<td>13797056</td>
<td>createFromPdu, getOriginatingAddress, getMessageBody</td>
</tr>
<tr>
<td>25992190</td>
<td>isScreenOn</td>
</tr>
<tr>
<td>22207722</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>20937784</td>
<td>getIntent</td>
</tr>
<tr>
<td>22704703</td>
<td>getIntent</td>
</tr>
<tr>
<td>7713222</td>
<td>create</td>
</tr>
<tr>
<td>11786050</td>
<td>create</td>
</tr>
<tr>
<td>8344528</td>
<td>addPreferencesFromResource</td>
</tr>
<tr>
<td>22999720</td>
<td>getIntent</td>
</tr>
<tr>
<td>11656197</td>
<td>removeTab</td>
</tr>
<tr>
<td>11652634</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>9967245</td>
<td>getIntent</td>
</tr>
<tr>
<td>28400191</td>
<td>getIntent</td>
</tr>
<tr>
<td>9697690</td>
<td>getDefault, sendTextMessage</td>
</tr>
<tr>
<td>3953791</td>
<td>setPictureListener, onNewPicture</td>
</tr>
<tr>
<td>13048916</td>
<td>getAuthToken</td>
</tr>
<tr>
<td>10913536</td>
<td>getIntent</td>
</tr>
<tr>
<td>11815354</td>
<td>getIntent</td>
</tr>
<tr>
<td>6134001</td>
<td>drawBitmap</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Question ID</th>
<th>Candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>24972742</td>
<td>getIntent</td>
</tr>
<tr>
<td>15782180</td>
<td>getIntent</td>
</tr>
<tr>
<td>23075051</td>
<td>getIntent</td>
</tr>
<tr>
<td>20388875</td>
<td>capturePicture</td>
</tr>
<tr>
<td>6264809</td>
<td>onStart</td>
</tr>
<tr>
<td>24703035</td>
<td>create</td>
</tr>
<tr>
<td>11944567</td>
<td>createFromPdu, getOriginatingAddress</td>
</tr>
<tr>
<td>23799438</td>
<td>getWidth, drawBitmap, getHeight</td>
</tr>
<tr>
<td>14093404</td>
<td>setLatestEventInfo</td>
</tr>
<tr>
<td>12474667</td>
<td>getIntent</td>
</tr>
<tr>
<td>5710304</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>7779110</td>
<td>create</td>
</tr>
<tr>
<td>17315269</td>
<td>create</td>
</tr>
<tr>
<td>25287270</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>6100010</td>
<td>registerListener</td>
</tr>
<tr>
<td>18436703</td>
<td>createFromAsset</td>
</tr>
<tr>
<td>12257459</td>
<td>addPreferencesFromResource</td>
</tr>
<tr>
<td>25652899</td>
<td>addTab</td>
</tr>
<tr>
<td>12563687</td>
<td>create</td>
</tr>
<tr>
<td>8866427</td>
<td>unregisterListener, registerListener</td>
</tr>
<tr>
<td>9766263</td>
<td>managedQuery</td>
</tr>
<tr>
<td>26165006</td>
<td>drawBitmap</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Question ID</th>
<th>Candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>12164024</td>
<td>getAuthToken</td>
</tr>
<tr>
<td>5752529</td>
<td>getIntent</td>
</tr>
<tr>
<td>16260622</td>
<td>speak</td>
</tr>
<tr>
<td>20625545</td>
<td>getIntent</td>
</tr>
<tr>
<td>16883811</td>
<td>create</td>
</tr>
<tr>
<td>18320983</td>
<td>setAlpha</td>
</tr>
<tr>
<td>19907116</td>
<td>getIntent</td>
</tr>
<tr>
<td>7740687</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>7175018</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>4182299</td>
<td>getIntent</td>
</tr>
<tr>
<td>5368774</td>
<td>drawBitmap</td>
</tr>
<tr>
<td>13365724</td>
<td>getIntent</td>
</tr>
<tr>
<td>15901775</td>
<td>create</td>
</tr>
<tr>
<td>9446206</td>
<td>create</td>
</tr>
<tr>
<td>20494849</td>
<td>setNavigationMode</td>
</tr>
<tr>
<td>18185828</td>
<td>createFromPdu, getOriginatingAddress,</td>
</tr>
<tr>
<td></td>
<td>getDisplayMessageBody</td>
</tr>
<tr>
<td>17475316</td>
<td>setTicker</td>
</tr>
<tr>
<td>22603153</td>
<td>setBackgroundDrawable</td>
</tr>
<tr>
<td>14683559</td>
<td>synthesizeToFile</td>
</tr>
<tr>
<td>8213477</td>
<td>getDefault, sendTextMessage</td>
</tr>
<tr>
<td>27372673</td>
<td>getIntent</td>
</tr>
</tbody>
</table>

Continued on next page
Table H.1—continued from previous page

<table>
<thead>
<tr>
<th>Question ID</th>
<th>Candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>12191204</td>
<td><code>createFromPdu</code>, <code>getOriginatingAddress</code>, <code>getMessageBody</code></td>
</tr>
<tr>
<td>20431451</td>
<td><code>createFromAsset</code></td>
</tr>
<tr>
<td>14472991</td>
<td><code>create</code>, <code>setButton2</code></td>
</tr>
<tr>
<td>7508185</td>
<td><code>create</code></td>
</tr>
<tr>
<td>8045556</td>
<td><code>setBackgroundDrawable</code></td>
</tr>
<tr>
<td>8740968</td>
<td><code>removeGlobalOnLayoutListener</code></td>
</tr>
<tr>
<td>5007318</td>
<td><code>getIntent</code></td>
</tr>
<tr>
<td>17629379</td>
<td><code>getIntent</code></td>
</tr>
<tr>
<td>14227023</td>
<td><code>create</code></td>
</tr>
<tr>
<td>21914978</td>
<td><code>drawBitmap</code></td>
</tr>
<tr>
<td>20642153</td>
<td><code>getAuthToken</code></td>
</tr>
<tr>
<td>24687087</td>
<td><code>addTab</code>, <code>newTab</code>, <code>setNavigationMode</code>,</td>
</tr>
<tr>
<td></td>
<td><code>setSelectedNavigationItem</code></td>
</tr>
<tr>
<td>21721247</td>
<td><code>drawBitmap</code></td>
</tr>
</tbody>
</table>
Bibliography


Bennet P. Lientz and E. Burton Swanson. *Software maintenance management: A study of*
the maintenance of computer application software in 487 data processing organizations. Addison-Wesley, 1980.


