Exploring the Use of Automated API Migrating Techniques in Practice: An Experience Report on Android

Maxime Lamothe, Weiyi Shang
Department of Computer Science and Software Engineering
Concordia University, Montreal, Quebec, Canada
{max_lam,shang}@encs.concordia.ca

ABSTRACT

In recent years, open source software libraries have allowed developers to build robust applications by consuming freely available application program interfaces (API). However, when these APIs evolve, consumers are left with the difficult task of migration. Studies on API migration often assume that software documentation lacks explicit information for migration guidance and is impractical for API consumers. Past research has shown that it is possible to present migration suggestions based on historical code-change information. On the other hand, research approaches with optimistic views of documentation have also observed positive results. Yet, the assumptions made by prior approaches have not been evaluated on large scale practical systems, leading to a need to affirm their validity. This paper reports our recent practical experience migrating the use of Android APIs in FDroid apps when leveraging approaches based on documentation and historical code changes. Our experiences suggest that migration through historical code-changes presents various challenges and that API documentation is undervalued. In particular, the majority of migrations from removed or deprecated Android APIs to newly added APIs can be suggested by a simple keyword search in the documentation. More importantly, during our practice, we experienced that the challenges of API migration lie beyond migration suggestions, in aspects such as coping with parameter type changes in new API. Future research may aim to design automated approaches to address the challenges that are documented in this experience report.

CCS CONCEPTS

• **Software and its engineering → Software evolution**;

KEYWORDS

Android API, API migration, Mining Software Repositories, Software evolution

ACM Reference Format:
Maxime Lamothe, Weiyi Shang. 2018. Exploring the Use of Automated API Migrating Techniques in Practice: An Experience Report on Android. In *MSR '18: MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3196398.3196420

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196420

1 INTRODUCTION

The current trends of mobile computing and software as a service present increasing opportunities for developers to rely on externally maintained software rather than consume their valuable development time [30]. However, as a consequence, software developers become dependent on frameworks and public application program interfaces (APIs) when developing their applications [14, 35, 42, 60]. When programming with an API, consumers must either use available documentation or code examples, in order to guide them in consuming the targeted API [32, 42].

As of 2016, the Google Play application store presents over 2.2 million applications [2]. All of these applications rely on the Android API to access device information and drivers. Released in September 2008 [40], the Android API is currently in its 26th version. This API provides a large number of varied functionalities for its consumers exposing more than 19,000 public methods. With over 1.5 million daily activations of Android devices, the use of the Android API is expected to keep growing in the coming decade [1].

Since the development of the API is typically independent from the consumption of the API, the consumers are at the mercy of the evolution of the interface. Prior research has concentrated on recommending or producing specialized tools to provide suggestions for consumers pursuing API migrations [7, 10, 14, 16, 21, 28, 33, 34, 39, 43, 44, 48, 59, 63]. These tools use various means, such as code documentation [6, 59] and historical code-change information when producing API migration suggestions [16, 20, 26]. Yet there exists no large-scale study to assess the effectiveness of these approaches in real-world API migrations. The popularity and the importance of the Android API makes it an ideal subject to conduct such a study.

In this paper we report our experiences with Android API migration using strategies described by prior research, namely those based on documentation and historical code-changes. Our findings are summarized in Table 1. As a first step, we opt to leverage the Android documentation [4], due to the important role of documentation claimed by prior research [6, 8, 9, 11, 12, 15, 18, 37, 38, 45, 51, 57, 62]. We find that although not all migrated methods can be found in the official Android online documentation, information needed to assist in API migration can be also found in other forms of documentation, such as code commit messages and code comments. Still, the official documentation contributes the majority (75.3%) of the information to suggest API migrations.

Table 1: Our Findings and Implications on Android API Migrations

Leveraging documentation (Section 3.1)	**Implications**
1) For 26% of the deleted or deprecated Android API methods, we could not find any replacements in our manual examination of the API.	Developers of migration tools and API consumers should be aware that not all modified methods have migration pathways. This might mean supporting an old version of the API or changing functionality.
2) Android documentation, including the online documentation, code comments, and commit messages often contains useful textual information for method migrations as well as information for their deprecation, addition, and removal per API version.	Android app developers should leverage such effective documentation as it allows them to understand and plan their API uses and migrations around the modifications to the API.
3) Migration pathways in documentation are often very explicit. Links between methods are clearly stated, and replacements are identified with complete method signatures for easy recognition.	Android app developers could recover Android API migration links by simple keyword search, instead of exploiting sophisticated techniques.
4) The Android official documentation [4] effectively presents migration pathways. Based on documentation alone, with naive text matching, we were able to automatically determine most of our manually identified migration paths.	Due to the high quality and the ease of access of the Android documentation, suggesting Android API migrations may not be a challenging task. Instead, migration research should concentrate on other tasks, such as handling different migration types.
Leveraging historical code changes (Section 3.2)	**Implications**
5) Historical code data, such as commits, only yields a few undocumented migration pathways and a fraction of migration pathways contained in documentation.	API migration researchers should employ historical code data as a backup when documentation is lacking, and not as a primary migration pathway source.
6) Some assumptions of history-based automated API migration are not met for the Android API, since a replacement method can be introduced earlier or later than the existing method, with a large time gap.	Android app developers should verify the assumptions of automated migration tools before exploiting them in practice.
API migrations in FDroid apps (Section 3.3)	**Implications**
7) Actual modified API usage is heavily centred around a few API calls. Most API users only require support for few modified API methods.	Android app developers and Android API architects could mine API usage data to prioritize their migration efforts.
8) API migrations often require further code modifications than simple renames or parameter changes, e.g. object instantiation.	Future research on API migration should investigate automated support to suggest code modifications examples for API migration.

In the second step, we leverage historical code change information (e.g. commits) to improve the results of the API migration suggestions from the previous step. In particular, prior approaches [16, 49, 59] that are based on this information typically assume that code change information is in the source code commits, i.e. if a method is removed, a replacement method should be added promptly. Therefore, we first examined this assumption in order to understand whether the techniques that are proposed in prior research can be leveraged in migrating APIs in practice. We found that most of removed/deprecated methods and newly introduced API methods for migration do not change in the same code commit. 30.4% of the new Android API methods are not even introduced in the same version as the removed/deprecated Android API methods. Furthermore, historical code change information only provides 42.7% of the necessary migration suggestions, and 90.5% of those are already indicated in the documentation.

To test the effectiveness of identified migration pathways, in the third step, we leveraged the API migration suggestions that we automatically recovered from both documentation (including official online documentation, commit message, and code comments) and historical code change information for FDroid apps.[1] We experience that only a small subset of the removed/deprecated API methods and API migrations are used by FDroid apps.

[1]Our automated script to recover Android API migration pathways is hosted online at https://lamothemax.github.io/MSR_2018_Android_API_Study/

Our results and experiences imply that even though documentation is often reported as incomplete or outdated [17, 60], developers should still consider the official documentation of the Android API as their major source of information. Moreover, before using any sophisticated techniques for API migration, developers should first verify the assumptions of those techniques before exploiting them in practice. On the other hand, developers could reduce and prioritize their efforts to a small subset of API methods, which are used in practice.

Our experience agrees with prior research [6, 17, 18, 26, 43, 44, 55, 59, 60] and shows that it is feasible to provide suggestions when migrating API methods to new versions. However, more importantly, after we successfully performed API migrations on three apps from one version of Android to the next, we found that implementing API migration code changes is much more challenging than identifying a migration pathway. Challenges such as migrating multiple related-APIs as well as changing object types present changes which would often require extensive knowledge and effort. We document these challenges in this experience report so that further research on API migration can investigate and propose automated solutions to assist API migration in practice.

The contributions of this paper include:

- We evaluate the use of documentation and historical code change information in API migration in a large scale subject.

- We find that the information needed to identify replacement API methods for migrations often resides explicitly in online documentation and repository commits as natural language text.
- We find that prior research-based sophisticated migration techniques may fail because particular assumptions are not met in practice.
- We documented our experiences and the challenges that we encountered when migrating the use of Android APIs in FDroid apps to benefit both practitioners and researchers.
- We documented the solutions we employed for our challenges, and presented our unsolved challenges as open challenges for future research.

The rest of the paper is organized as follows: Section 2 provides a background on API migration practices and past research. Section 3 presents the methodology followed in our study and reports our experiences the challenges we encountered. Section 4 discusses related work. Finally sections 5 and 6 outline threats to validity and the conclusions of our paper.

2 ANDROID API MIGRATION

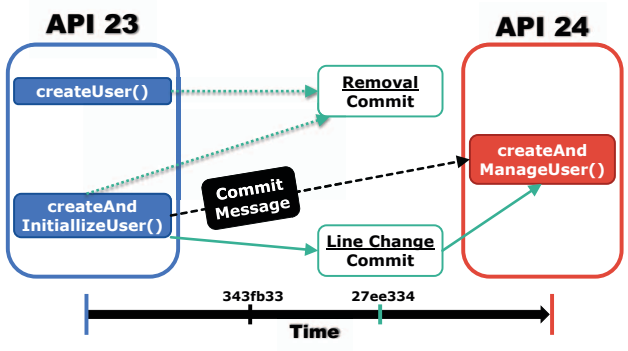
Android app development heavily depends on the availability of Android APIs. In the most recent version of Android, "Oreo", there currently exist 3,354 API classes and 33,560 API methods. However, such APIs are updated every 6 to 12 months when Android releases a new version. A prior study by McDonnell et al. showed that on average 115 Android APIs are updated per month [32]. Such API updates would cause around 28% of API references to become outdated in a median lagging time of 16 months, while upgrading these updated APIs takes about 14 months [32]. Prior research shows that Android app developers seek to update their API usage, however their migrations are slower than the API updates [32]. Therefore, efficient migration techniques are essential to help Android app developers.

Android architects maintain an open online documentation to share information on available API and to communicate API deprecations in each version [4]. However, knowing that an API is deprecated may not give the consumer enough information on the existence of a replacement API or which new API is needed to replace any loss in functionality.

For example, the Android API has been releasing new versions since September 23, 2008 [40]. The Android project provides a number of resources to help consumers keep track of changes in the API. However, even with its well maintained documentation, it is sometimes required for an API consumer to look at the API source code to determine how to migrate a removed API method as presented in Section 2.1.

In the Android project, behaviour changes of API methods are sometimes documented and presented with new API releases. This documentation can be used to locate method substitutions directly, and this has been used in this research in order to check the results of our links. However, not all versions of the API provide this documentation. We assume this documentation is rarely done due to the resource requirements of maintaining such a list. Having a tool to do this automatically, or at the very least to check the list for errors, would be a welcome boon for maintenance efforts.

Figure 1: Example of methods that are linked through commit history.



2.1 A real-life example

API Documentation alone is sometimes insufficient to determine the migration of a removed method. For instance, the API method `createAndInitiallizeUser` from `android.os.UserManager` was removed between Android API versions 23 and 24. The method was replaced by a new method, `createAndManageUser`. Since the method was removed, no information on the methods is henceforth available in the most recent Android official online documentation. However, it is possible to find the removal of the method in the change documentation of the API [4].

In order to determine the genuine evolution of the method, one needs to look at the revision history of the framework. Due to the open source nature of the Android framework, the API version control repository is available online. Official online documentation of the `createAndInitiallizeUser` method does not provide useful information for the migration. However, by looking at repository commits, it is possible to see that the `createAndInitiallizeUser` method was replaced by the `createAndManageUser` method in commit *343fb33* as shown in Figure 1. The information can also be found in the internal code comments and commit messages. Moreover, by looking at other commits during the history of the Android version, we find another method `createUser` that is co-changed with `createAndInitiallizeUser`, while having no other documentation about the removal. This requires creativity and research on the part of the user in order to find a substitute. However, by looking at the commit history, and carefully parsing the commit comments, it is possible to determine that the `createAndInitiallizeUser` and `createUser` are interlinked through removals, modifications, comments, and Java class, but they are never explicitly linked. Therefore, a user that wishes to update their use of the `createUser` method should also take a careful look at the `createAndInitiallizeUser`.

This example is particularly interesting because it shows that:

- Not all methods that have replacement methods present the information in official documentation.
- Consumers of methods such as these are expected to put in the effort to find the replacement themselves.
- API migrations can involve multiple API methods.

Examples such as these are the primary motivations for this work. Ideally, with a complete mapping of all methods and their relationships to other methods, developers should be able to get

an understanding of migrations with a simple glance. Therefore, in this paper, we aim to evaluate the applicability and usefulness of existing approaches on automated Android API migration.

3 AN EXPERIENCE REPORT

In this section, we aim to explore the use of existing automated API upgrading techniques to migrate the Android API. In particular, we explore the use of documentation and historical code change information based techniques to assist in migrating removed or deprecated Android API methods.

In order to evaluate the use of existing automated API migration techniques, we first need to extract all the changes to Android API including additions, deletions, deprecations, or modifications to existing API in each version of Android. In particular, we select the most recent six versions of Android (21 to 26). We first leverage JDiff to identify all added, removed, deprecated and modified APIs between every two Android versions. The Android API changes are summarized in Table 2. In particular, we consider the total amount of removed or deprecated API as the upper bound of all possible API migrations, since they would suggest or even force developers to change their source code in order to adapt to new APIs.

Figure 2: API migration extraction strategies.

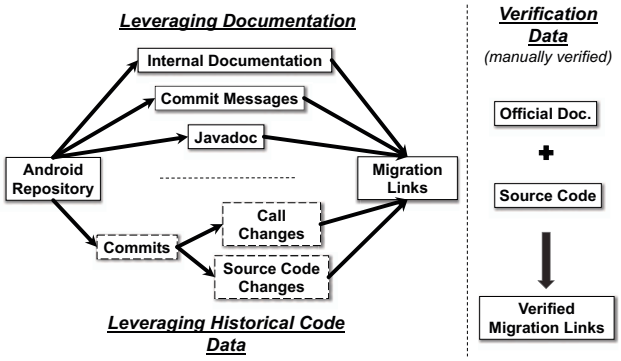


Table 2: Android API Modifications per API Version

		Class			Method			Field		
API Version	**Release Date**	Added	Changed	Removed	Added	Changed	Removed	Added	Changed	Removed
16	2012-07-09	57	211	0	381	151	20	171	46	4
17	2012-11-13	41	111	2	150	37	19	155	69	3
18	2013-07-24	61	108	36	155	44	4	131	25	1
19	2013-10-31	78	180	0	268	26	5	391	6	0
20	2014-06-25	10	25	0	32	5	1	45	2	0
21	2014-11-12	147	360	0	770	117	29	1150	75	2
22	2015-03-09	4	439	0	73	128	3	53	1	13
23	2015-10-05	119	257	36	541	132	38	466	89	83
24	2016-08-22	147	433	3	877	127	13	585	31	5
25	2016-10-04	5	38	0	50	1	0	53	0	0
26	2017-08-21	145	349	4	795	139	18	572	69	0
Min		4	25	0	32	1	0	45	0	0
Max		147	439	36	877	151	38	1150	89	83
Mean		**57**	**155**	**4**	**266**	**58**	**10**	**254**	**35**	**8**

For the purposes of this study we concentrated on the six most recent versions (API Versions: 21-26)

In the rest of this section, we report our experiences during the migration of APIs uses in Android apps. We discuss our experiences in three steps. For each step, we discuss its motivation, our approach, and the outcome of the step as well as the challenges that we faced.

3.1 Step 1: Leveraging documentation in API migrations

Motivation

Due to the high dependence between Android apps and Android APIs, ideally, all removed and deprecated APIs should be properly documented, such that consumers can opt to adopt other APIs to sustain the functionality of their apps. In addition, previous research has shown that documentation can be used to determine migration pathways in changing API [59]. Therefore, in this step, we seek to determine whether the Android API documentation can be leveraged when assisting with API migrations.

Approach

In order to automatically recover API migration suggestions from documentation, we consider three readily available sources of data as documentation: 1) code comments in JavaDoc format in the source code, before the declaration of each method, 2) code commit messages and 3) official Android online API documentation.

Code comments in JavaDoc. We first obtain all the source code for each version of Android. We then use srcML [13] coupled with python scripts to extract all the JavaDoc code comment for each Android API. For each code comment, we use the API name as keywords, and automatically search whether the name of a changed (added, deleted, or deprecated) API is mentioned in the comment. For example, as shown in Figure 3, in API version 23, `android.content.res .Resources.getColor(int)` was deprecated and obtained a JavaDoc link to its migrated method: `android.content .res.Resources.getColor(int, Theme)`.

Figure 3: `getColor(int)` source code snippet presents a migration pathway.

```
 * @return A single color value in the form 0xAARRGGBB.
 * @deprecated Use {@link #getColor(int, Theme)} instead.
 */
@ColorInt
@Deprecated
public int getColor(@ColorRes int id) throws NotFoundException {
    return getColor(id, null);
}
```

Code commit messages. We extract all code commits and their commit messages between every two consecutive versions of Android from the *git* repository [5]. Similarly to code comments, we automatically search whether the name of a changed API is mentioned in the code commit message. For example, as presented in section 2.1, and in Figure 4, `createAndInitiallizeUser` presented a link to `createAndManageUser` in commit message 343fb33.

Official Android API documentation. The Android API documentation contains a list of added, deleted, or deprecated APIs in each version [4]. By checking the online documentation of each deleted or deprecated API, we manually examine whether the official documentation provides a replacement for the deleted or deprecated APIs. For example, `android.text.Html.fromHtml(String)` appears in the online documentation as shown in Figure 5. It is also possible to mine the documentation from JavaDoc links in the historical code-data information. The `createAndInitiallizeUser` method and its migration also appeared in online documentation, however only in the framework repository documentation [5].

Results

Figure 4: Android framework commit message 343fb33, presents a migration pathway.

Add new API function createAndManageUser

This is a reduced version of the (deprecated) function **createAndInitializeUser**, that allows the device owner to create a new user and pass a bundle with information for initialization. The new version of the function has the same functionality, but the profile owner of the new user is always the device owner.

A flag can be specified to skip the setup wizard for the new user.

The new user is not started in the background, as opposed to how **createAndInitializeUser** did it. Instead, the bundle with initialization information is stored and will be broadcast when the user is started for the first time.

Bug: 25288732, 25860170
Change-Id: I4e1aea6d2b7821b412c131e88454dff5934192aa

Figure 5: Android online documentation for method `fromHtml`, presents a migration pathway.

The majority of replacements for deleted or deprecated APIs can be recovered from the explicit wording in documentation. For the six studied versions, we were able to determine between 51% and 98.4% of the deleted or deprecated APIs through documented replacements. We then manually examined the APIs for which we could not recover a replacement, in order to understand whether those APIs do not have an replacement or whether we missed a documented replacement. For 26% of the deleted or deprecated APIs, we cannot find a replacement at all (possible removal of functionality). For example, all of the methods present in the `PskKeyManager` class were removed without replacement when the class was removed due to incompatibilities with TLS 1.3.

Experience #1: For 26% of the deleted or deprecated Android API methods, we could not find any replacements in our manual examination of the API.

We also found an extra 21.5% of replacement APIs which exist, but we were unable to recover them explicitly from the documentation. Some knowledge of the project had to be combined with the documentation. For example, if method functionality had been migrated to a different class, which existed prior to the studied release, we were unable to provide a replacement automatically. This was particularly prevalent when methods migrated from using static methods to classes which produced similar results through new objects. Externally maintained replacements as part of the `java.lang.Math` package account for 8 of the 29 replacements which were not found for API version 23. One of the undiscovered replacements was a Java wrapped C method which proved difficult to link. However, we were able to link a similar method in API version 24 through its documentation links. Out of the 15 other undiscovered replacements, all of them referred to another class to replace the lost functionality.

Experience #2: Android documentation, including the online documentation, code comments, and commit messages often contains useful textual information for method migrations as well as information for their deprecation, addition, and removal per API version.

Experience #3: Migration pathways of APIs in documentation are often very explicit. Links between methods are clearly stated in the documentation, and replacements are identified with complete method signatures for easy recognition.

The official documentation of Android API is the main source of data for suggesting API migrations. Android provides a rich documentation from the official documentation website [4], and from the framework commits [5]. We find that the documentation of Android API provides more migration links than any other sources. By manually inspecting all the sources of information of the removed or deprecated APIs, we find that only 5 out of our 469 studied APIs had migration paths that were not presented in the official online documentation. However, as presented in challenge #2, not all previous documentation is readily available in the latest online index [4], and some of it must be mined from the JavaDoc in the repository [5].

Our results show that identifying replacements for the removed or deprecated Android APIs may not compose a challenging task, since developers may not need sophisticated techniques to analyze documentation in order to detect the API replacement, while simple keyword searching on the API names may recover the majority of the API replacement. More importantly, the majority of the replacements can be recovered from the official documentation. Compared with the code comments and the commit messages, the online Android official documentation is the easiest to access and to analyze by consumers. This finding also implies that developers may not need sophisticated techniques nor access to the software repositories to migrate Android APIs.

Experience #4: The Android official documentation [4] effectively presents migration pathways. Based on documentation alone, with simple text matching, we were able to automatically determine most of our manually identified migration paths.

Challenges

Challenge #1: Associating documentation to APIs.

Description: Source code documentation is not always favourably located in order to determine the targeted source code artifacts. During this research we noticed that sometimes documentation provided at the top of a Java class can give migration or removal information about a method in the class. However, linking this documentation requires foresight of its existence.

Our solution: Since our method of using naive text matching worked well for migration suggestions, we determined that in the case of the Android API we could also apply text matching to documentation found throughout a given class (cf. Table 3). After identifying the removal of a method in a given API we suggest looking at all unlabelled documentation in its class for text matches, and attempting to identify any other method mentioned.

Challenge #2: Missing historical information of API documentation.

Description: All references to removed methods are expunged in the official Android documentation. Therefore when a method finally gets removed, it is no longer possible to find its information on the Android developer website [4]. Likewise, the JavaDoc for the project is not provided for removed API methods. This likely prevents the misuse of inaccessible methods, however it makes it more challenging to find migration paths for removed methods.

Our solution: Although the documentation for removed methods is not directly accessible from the Android developer site, it is accessible in the source code repositories JavaDoc. Therefore, it is possible to mine the source code history for documentation information which was removed, in order to build a complete migration picture. It could help slow adopters if Android built this information into the website as a removed section to help them migrate very old app versions.

Open Challenge #1: N-to-N API methods migrations

Description: Using the current approaches, it is difficult to assist in the migration between two sets of multiple APIs as a whole, i.e., N-to-N migration scenarios. First of all, with current techniques it is difficult to determine if a migration path search is returning multiple results because of false positives or due to multiple migration paths. Secondly, understanding the relationship between the multiple APIs is challenging. Current approaches concentrate on one-to-one migration scenarios and shy away from automatically creating new source code that consists of multiple new migrated APIs.

Table 3: Android API suggestions automatically found, compared to manually confirmed migrations.

Only methods with replacements are presented here.

API Version	Found Replacement	Missed Replacement
22	5	1
23	31	29
24	25	3
25	1	0
26	62	1

3.2 Step 2: Leveraging historical code-change information in API migrations

Motivation

In the previous research step, we found that a large portion of API upgrades can be recovered by searching simple keywords in documentation; only 4% of API upgrades were unrecoverable by only analyzing documentation. On the other hand, prior studies leverage software development history, such as code change per commit, when assisting in recovering API migrations [16, 17]. For example, SemiDiff identified code changes within a commit to determine API method replacements [16, 17].

We do not directly test any specific tool as many of them have not been maintained, or require modification to run on our chosen project. Since modifying the tools could introduce errors or a bias for certain methods, we chose to test the underlying assumptions of API migration techniques in an effort to determine whether these underlying assumptions and theories hold in practice.

These techniques often assume that the removal of an existing API and the addition of an upgraded APIs exist within a short period of time (i.e. within a few commits)[16, 20, 26, 42]. Since such an assumption is heavily depended upon, yet never validated in practice, the assumption can lead to uncertainly in the usefulness of automated API upgrading techniques. Therefore, we aim to leverage historical information to recover Android API upgrades.

Approach

We first leverage code change history in the implementation of the removed or deprecated API methods. If two methods change implementation in the same commit, it is likely that their implementations are linked in some way. The more often two methods present simultaneous implementation changes, the more likely they are to share implementation details. This can allow us to determine which methods provide similar features and make links between features that would not be available by looking at release snapshots.

We collect all commits in the *git* repository of Android. For each commit, we identify the Android APIs that are changed. Since *git diff* would only provide textual based differences in a commit, we use *srcML* [13] as an intermediary to provide XML representations of Abstract Syntax Tree (AST) of the source code. By comparing *srcML* output of each source code file before and after a commit, we are able to identify which method is changed in the commit. We then track all API implementations that are co-changed with the API implementation that is removed, modified or deprecated in the Android release. Although not all co-changes present migrations, most, if not all, migrations should present co-changes. We study whether these co-changed APIs can provide useful information for recovering API upgrades [41].

Second, for each of the known API upgrades (see Section 3.1), we examined the time span between its deprecation (if present), the removal of the existing API and the introduction of a new API.

Results

Over all the Android API versions studied, source code change history provides a total of 53 migration pathways. Out of these pathways, only 5 are uniquely identified by commit information. However, documentation with basic text matching identifies 119 suggestions. The Android API documentation suggestions include 90.5% of the migrations found through source code change history.

> Experience #5: Historical code data, such as commits, only yields a few undocumented migration pathways and a fraction of migration pathways contained in documentation.

Existing APIs are not always deprecated, removed, or modified in the same commit as new APIs are introduced. Based on our manually identified replacements, we found that for 57.3%

of them, we could not identify any commit migration pathways between the outgoing API and any replacement API.

Newly introduced APIs are often added into source code earlier than the removal or the deprecation of the existing APIs. Table 4 presents the API version difference between the appearance of a replacement method and the removal, deprecation, or change of the original method. In the studied system, 59.5% of modified methods have a replacement which appears in the same version as the modification. 10.1% of modifications have replacements outside of the Android API, and the rest of replacements are spread over the entire evolution of the API. For example, the method `getCellLocation()` was deprecated in API version 26, and was given a documented migration pathway to API method `getAllCellInfo()`. However, `getAllCellInfo()` was introduced in API version 17. Therefore, no clear migration pathway exists in API version 26 other than documentation.

In three cases, the Android API method replacement was provided in future releases. This makes it impossible to determine a replacement functionality at deprecation time for these methods, as it does not exist yet. It also makes it impossible to use commit based links since the methods clearly are modified in different releases. However, with constant monitoring of the project, it may be possible to determine a replacement through documentation and commit messages. Similarly, migration paths that appear multiple releases before deprecation time, may not be linkable through commits. Therefore we must depend on documentation to tell us when links are created.

There are many deprecated methods left in the source code without removal. In the Android API versions studied, deprecated APIs outnumber removed APIs by a factor of 2.94. Through our research of migration methods and their emergence, we determined that there are more deprecations (244) than removed (83) and changed (142) methods in the versions studied. This presents us with an interesting finding. Only a fraction of deprecated methods were removed. This presents a contrast to Zhou and Walker [64] who show that removed API outnumber deprecated API significantly. This is not the case for the most recent Android versions. We did notice that some methods were undeprecated, such as `android.app.Notification.Builder.SetNumber(int)`, however only a few such outliers were found in the versions studied.

> Experience #6: Some assumptions of history-based automated API migration are not met for the Android API, since a replacement method can be introduced earlier or later than the existing method, with a large time gap.

Challenges

Open Challenge #2: Identify the time gap between the addition and removal of APIs

Description: Our findings indicate that many API methods use migration paths that are introduced in a different version than the deprecation or removal of the targeted method. This makes it difficult to use commit based methods to identify a migration path between two methods. Table 4 shows that a large amount of modified methods have a replacement introduced earlier than their removal/deprecation. By widening the search for a migration path to a wider release cycle, it may be possible to identify these migration paths without documentation. However our experience shows that widening the search increases the amount of false positives. Therefore, we believe developers should minimize the use of broad time-spans when searching for method replacements and instead determine ways to optimize their historical data search through documentation informed time spans.

Figure 6: API Migration Mapping Example

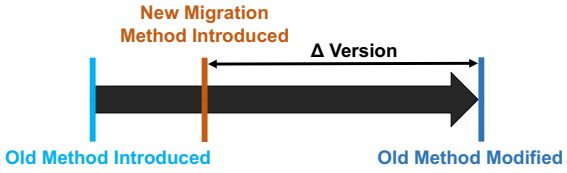


Table 4: API migration mapping version Δ
Negative Δ implies migration introduction before modification.
Positive Δ implies migration introduction after modification.

Δ Version	Modified Methods
-25	11
-23	1
-22	2
-18	7
-14	1
-13	3
-9	1
-8	2
-6	1
-5	8
-3	2
-2	2
-1	4
0	94
1	2
3	1

3.3 Step 3: API migration in FDroid apps

Motivation

In the previous research steps, we recovered API migration information using both documentation and historical code change information. However, such information may not be enough nor beneficial at all when migrating API usage in real-life Android apps. Therefore, in this step, we seek to determine how these links should be used to facilitate API usage in open-source Android apps from FDroid [3].

Approach

Android API usage in FDroid Applications

FDroid Android applications are open-source Java applications that call the Android API to interact with an Android end user. Prior studies have been done on the FDroid dataset [25]. With knowledge of the available Android API methods, it is possible to mine the FDroid applications for their API uses. We first mined the FDroid

database for FDroid projects which had multiple versions and had downloadable source code. We then used a list of removed Android methods to determine the usage of removed Android methods in these FDroid projects. The list of removed methods is maintained by the Android project as part of their public framework repository [5]. Finding the usage of removed methods is done by parsing all files in all 415 FDroid projects for uses of the removed methods. We counted how many times a removed method was used, which file it was used in, and in which version of each project. We then use this information to determine the popularity of removed methods, and to find which methods are preventing an app from targeting a higher API level.

To determine whether the links produced by our approach could be useful to developers, we looked at the links between removed methods and their replacements. We gathered the uses of removed methods from a sample of 415 open source Android applications. We concentrated on Android API versions 21-25, as 66.4% of applications targeted these versions. Although Android API version 26 was used during this study, we did not have any FDroid projects with uses of the API and it is therefore not present in this research step. Since Android API version 25 only deprecated one method we do not present data related to it for clarity.

Migrating API usage in FDroid Applications

To test our suggestions in a more rigorous fashion, we use three applications which are blocked from changing API versions due to removed methods. Using our list of FDroid application method uses, we identified three applications, *Tasks*, *Forrunners*, and *Poet-Assistant* to test our migration suggestions and attempt to migrate the applications from one version API version to the next. These applications were chosen because they presented multiple app releases (6-167), they were prevented from accessing Android API 24 due to their use of methods which were removed after API 23, and had included test suites in their development packages. We manually migrated the apps by using our suggested migration methods and ran their test suites to see if any tests were broken by our changes. We also successfully ran the apps in the Android Studio's development environment simulator as a safeguard against defective or lenient tests. We specifically attempted to target the modified functionality in the simulator, and did not experience any crashes.

Table 5: Android API methods found in FDroid projects.

API Version	Changed API Methods	Found in FDroid	Can Migrate
22	128	5	3
23	157	28	20
24	56	22	11

API levels 25 and 26 are not presented here for clarity.

Results

Only a small sample of APIs are used in FDroid projects and a small number of APIs account for the majority of the removed APIs. Not all methods which were removed by the Android development team were sampled in our study of 415 FDroid projects. Table 5 shows that between 4% and 39% of removed API methods were sampled in the FDroid projects. Therefore, not all removed methods bear the same migration weight. This implies that API architects can focus on a small amount of APIs to prioritize API migration efforts.

> Experience #7 Actual modified API usage is heavily centred around a few API calls. Most API users only require support for few modified API methods.

API migrations may vary in scope. Not all migrations are equal in scope [21]. In the versions of the Android API studied we found multiple migration types.

Some migrations require the removal of one or more parameters such as `android.hardware.usb. UsbRequest.queue(Bytebuffer, int)` which became `queue(Bytebuffer)` in API version 26. The removal of a parameter could mean reworking some code if old parameters were joined.

Other migrations require the addition of one or more parameters, such as `android.text.Html.fromHtml(String)` which was changed to `android.text.fromHtml(String, int)`. The int argument was added as a way to return different flags from the `fromHtml` method. If the user wanted to keep the same functionality as the previous version of the method, the int could simply be set to a value of 0. Therefore, although inconvenient, this change is relatively simple to develop.

Similarly, some methods were simply refactored to a different class, the `FrameLayout` class `getForeground()` methods were migrated to the `View` class. For methods like these, simple refactoring could allow these methods to be migrated [21]. This is not a problem as long as the classes were not delegated outside the Android repository which happened to the `FloatMath` Android expressions which were relegated to the `java.lang.Math` package.

Comparatively, the WebViewClient method `shouldOverrideUrlLoading(WebView, String)` changed parameter type and migrated to `shouldOverrideUrlLoading (WebView, WebResourceRequest)`, and contained a very different migration strategy. The API consumer now has to learn how the new WebResourceRequest works and properly instantiate the object. This is a slightly more difficult task for a developer. A machine would almost assuredly require code examples from which to map the changes. Giving the maximum amount of available information to the API consumer in these cases allows them to determine if the migration is worthwhile. If they determine that the migration cost is acceptable, they then have access to links which could help them understand the new functionality.

> Experience #8 API migrations often require further code modification than a simple rename or parameter change, e.g. object instantiation.

Developers may migrate APIs while keeping support for the old API version. In two cases, we identified applications which were using migration methods as expected and the migrations were done without any problems. However, in the case of *Poet-Assistant* we discovered that the developers were already aware of the migration of these methods. The developers had put in conditional statements to determine the API level of the user in order to determine which API call to make. Therefore, the developers had

two solutions in place for every use of the `fromHtml` method. this allowed us to determine that our suggested method was the appropriate migration method for this situation. It also presented new information that we had not anticipated. Some developers are willing to support multiple versions of the Android API simultaneously. Although *Poet-Assistant* developers had gone through the effort of making the migration to API 24, they had kept the backwards compatibility functionality for previous API levels.

Challenges

Challenge #3: Ranking migrations suggestions

Description: Using naive text matching alone, or historical source code data alone, provides multiple false positives, such as similar method names, for migration paths. Therefore, it is often challenging to rank migration suggestions in a way that makes them usable to an end user.

Our solution: By coupling both approaches we were able to produce more accurate migration suggestion rankings. We used both approaches independently and ranked migrations based on a coupled answer from both sources of information. More accurate methods for both documentation mining and source code mining could be developed, but we believe that coupling both sources of information will lead to more accurate results than having them separate.

Open challenge #3: Identifying the existence of API replacements.

For the Android API versions studied we found that many (66.3%) API modifications did not contain migrations. It is possible that we missed migration pathways in our manual examination. However, in practice it makes little difference if there is no replacement or if the replacement is too difficult to find. Either way the API consumer does not obtain a replacement and will assume one does not exist. Therefore, we open a challenge to develop an approach to determine a gold-standard to identify the existence of migration pathways. Currently, we have no way to be certain whether a method migration exists without documentation or architecture information, and it is possible for these sources of information to fail.

4 RELATED WORK

In this section, we discuss prior related research based on leading approaches for analyzing API method migrations.

4.1 Existing automated API migration techniques

Due to the importance and challenges of API migration, there exist prior research that proposes automated techniques to assist in API migration. In particular, these techniques are mainly either based on the development history or documentation of the API.

4.1.1 API migration techniques based on development history . Previous research has produced numerous API migration mapping methods that rely on historical source code repositories.

Dagenais and Robillard produced SemDiff, a tool for recommending adaptations to clients of a changing framework [16, 17]. By studying the changes to internal method calls of a framework, SemDiff produces a list of suggestions for migrations of framework methods. SemDiff uses repository information to observe changes in source code in order to create recommendations to method migrations.

Dig and Johnson [20] present an approach to update applications by using a tool to record refactorings done on a component application and then replay them on a target application. Their research proposes that this could be used to refactor API uses from one version to another. The approach presented requires two versions of an application to target in order to determine the refactoring that produces changes between the two versions.

However, since these methods rely on historical source-code information, they make the assumption that the information they seek can be retrieved from historical data. Furthermore, they have no way to determine if there is no answer to a migration pathway other than brute force. Likewise, if migration pathways are introduced over long periods of time, the pathways may not be found in the commits studied. Furthermore, these methods usually present the best migration pathway [16, 17] and may not present migration pathways that require multiple APIs.

Techniques such as those proposed by Nita and Notkin [44] and Wang et al. [60] use programmed guidelines to migrate from one API to another. The use of guided mappings allows full developer involvement and therefore testing of the migration. However, it is a resource intensive approach, since migration paths must be manually deliberated. Henkel and Diwan [26] produced an Eclipse plugin called CatchUp! which captures refactorings while an API architect is making modification to the API and replays the refactoring in a target application in order to migrate the API consuming application. These approaches currently require foresight from API developers since the techniques must be active during development as there is currently no tool to mine all of the required refactoring information from historical code data.

4.1.2 API migration techniques based on documentation. Tools such as JDiff [6] use Javadoc information to present differences between any two versions of documented Java software. As long as the software is produced in the Java programming language and properly documented, JDiff can tell a user whether a method has been changed, removed, or added based on a previous software version. JDiff converts source code to XML representations to determine which code items were modified. This information is then augmented through internal documentation based on Javadoc annotations [6]. Similarly, the JaSCUT Generator [59] parses source code to find Javadoc annotations, which can be used to heuristically find API mappings. We used a similar approach to track methods through commits in order to determine if any renames, removals, or additions to source code had taken place. JDiff concentrates on the addition, removal, or modification of source code items such as fields, methods, classes, and packages. However, other than these changes, it does not generate any links that do not exist in the Javadoc.

4.2 API Suggestions

Prior research has produced numerous API studies and API suggestion approaches [7, 10, 11, 14–39, 41–47, 50, 52–61, 63, 64].

These approaches could be used to help API consumers when attempting to find suggestions to migrate their apps. ApiRec produced by Nguyen et al. [42] uses statistical learning approaches

to correctly recommend API calls 77% of the time in their top 5 guesses. The system is based on the intuition that, developers make low-level changes while having a higher-level intent in mind. Similarly, Nguyen et al presented a vector representation code mapping approach also based on statistical approaches [43]. Using their vector mining approach they are able to mine API mappings between the Java JDK and the C# .NET framework. Their approach is based on mining usage relations and contextual uses of other APIs. Using this data they were able to detect pairs of API with different names which could allow mapping from one programming language to the next with an accuracy ranging from 42.8% to 73.3%. In this paper, instead of suggesting API usage, we focus on the use of documentation and historical code change information to automatically suggest API migration.

4.3 API Documentation enhancement

Several approaches to link documentation to source code have been explored in the past. Dagenais and Robillard presented an approach to automatically analyze documentation and link code elements to documentation terms [18]. Their prior experience showed that API consumers repeatedly asking questions led to the improvement of documentation. This led them to use documentation augment source code understanding. Similarly, Chen produced a system integrated into the Eclipse IDE to provide source code to documentation visualization links [12]. This approach uses a mix of information retrieval and text mining techniques to uncover links and then displays them using the eclipse visualization toolkit. ARENA, a tool proposed by Moreno et al. uses both source code analysis and documentation to produce release notes for API changes [37]. This type of tool can be directly useful to developers by documenting buried API links, and could potentially be modified to use the automated release notes for migration automation. The rich knowledge and value of documentation motivates the first step of our work and our results shows that Android documentation provides valuable information to assist in API migration.

5 THREAT TO VALIDITY

The following section aims to address the various threats to validity present in our research and how these problems were mitigated.

Construct validity.

It is possible that due to improper maintenance, documentation and source code are not representative of one another. Since all the information in this research was mined from documentation and source code history, this would cause the information in this project to be ineffective at showing the links between various methods. Any links created from unsynchronized documentation and code, would tend to arbitrary directions. We believe this to be unlikely for multiple reasons. The Android project is popular and used to support millions of apps. These millions of apps rely on and expect a high quality API as service to their businesses. It is likely that inconsistencies in documentation and source code are rapidly reported and fixed. The links produced in this study have been tested as migration links through manual inspection of both the Android framework and its documentation. No inconsistencies were noticed during this research. Our suggested method links were also tested on three different projects and the suggestions provided have been shown to compile and produce working applications.

External validity.

Since Android and Android applications were the only case studies done for this work, it is possible that the findings determined in this report are not common to other projects. Therefore these results might not generalize to other projects. We attempt to mitigate the drawbacks of this threat by making our findings as general as possible, and looking at general trends in our project sample.

We also cannot claim that the findings in this report can generalize to other programming languages. The study done in this report requires Java tooling and Java files. Therefore, it is possible that the findings in this report are only indicative of Java APIs. However, since documentation and source code repositories are not unique to the Java programming language, we believe that our findings have the potential to apply to APIs from any programming language.

Internal validity.

The findings in this report are all based on documentation that was accepted by the Android development team. This might present a bias, as it is possible that the Android development team uploads only documentation that is favourable to them and removes older documentation to hide any mistakes or risks caused by erroneous documentation. We did not observe documentation maintained outside of the source code repository. This was mitigated by looking at all available commits, which should present all changes in documentation. If any inconsistencies in documentation exist, they should appear in the committed changes to the documentation, and none were noticed. If the Android development team only presents filtered commits which have perfect documentation, we must accept the available information at face value as we have no other sources of information about the API.

6 CONCLUSION

In this paper we present our experience in using Android API source code repository coupled with documentation to provide suggestions for Android API migration. During our practice, we find that although a portion of the removed or deprecated API methods do not have a replacement, identifying a replacement using documentation or historical code change information is not a challenging task for practitioners. In particular, Android official online documentation provides valuable information and enables the use of simple keyword searches to find a replacement for removed or deprecated API methods. Existing tools such as ARENA [50] should theoretically be able to mine this information. However, when we applied API method replacements to migrate Android API methods in FDroidApps, we experienced other challenges, which are more time consuming to address, such as initializing new parameter types. We document these challenges so that future research can investigate them and propose automated techniques to assist in API migration.

This paper highlights some failings of current API migration techniques and provides opportunities to improve the understanding of API evolution, particularly API migration. Several challenges to be tackled by future research have been presented based on our experience with the Android API and FDroid apps.

REFERENCES

[1] 2017. Android. (2017). https://www.statista.com/topics/876/android/

[2] 2017. App stores: number of apps in leading app stores 2016. (2017). https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

[3] 2017. F-Droid. (2017). https://f-droid.org/

[4] 2017. Package Index. (Jul 2017). https://developer.android.com/reference/packages.html

[5] Android. 2017. Android Platform Frameworks Base. (Aug 2017). https://github.com/android/

[6] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2006. JDiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering* 14, 1 (2006), 3–36.

[7] Muhammad Asaduzzaman, Chanchal K. Roy, Samiul Monir, and Kevin A. Schneider. 2015. Exploring API method parameter recommendations. *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2015).

[8] Alberto Bacchelli, Michele Lanza, and Romain Robbes. 2010. Linking e-mails and source code artifacts. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE 10* (2010).

[9] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. 2017. On the Optimal Order of Reading Source Code Changes for Review. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2017).

[10] Raymond P. L. Buse and Westley Weimer. 2012. Synthesizing API usage examples. *2012 34th International Conference on Software Engineering (ICSE)* (2012).

[11] Wing-Kwan Chan, Hong Cheng, and David Lo. 2012. Searching connected API subgraph via text phrases. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE 12* (2012).

[12] Xiaofan Chen. 2010. Extraction and visualization of traceability relationships between documents and source code. *Proceedings of the IEEE/ACM international conference on Automated software engineering* (Sep 2010), 505–510.

[13] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. 2013. srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. *2013 IEEE International Conference on Software Maintenance* (2013).

[14] Bradley E. Cossette and Robert J. Walker. 2012. Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 55, 11 pages.

[15] D. Cubranic, G.c. Murphy, J. Singer, and K.s. Booth. 2005. Hipikat: a project memory for software development. *IEEE Transactions on Software Engineering* 31, 6 (2005), 446–465.

[16] B. Dagenais and Martin Robillard. 2008. Recommending adaptive changes for framework evolution. *Proceedings of the 13th international conference on Software engineering - ICSE '08* (2008).

[17] Barthélémy Dagenais and Martin P. Robillard. 2011. Recommending Adaptive Changes for Framework Evolution. *ACM Transactions on Software Engineering and Methodology* 20, 4 (2011), 1–35.

[18] Barthelemy Dagenais and Martin P. Robillard. 2012. Recovering traceability links between an API and its learning resources. *2012 34th International Conference on Software Engineering (ICSE)* (2012).

[19] Marco Dambros, Michele Lanza, Mircea Lungu, and Romain Robbes. 2009. Promises and perils of porting software visualization tools to the web. *2009 11th IEEE International Symposium on Web Systems Evolution* (2009).

[20] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. Automated Detection of Refactorings in Evolving Components. *ECOOP 2006 – Object-Oriented Programming Lecture Notes in Computer Science* (2006), 404–428.

[21] Danny Dig and Ralph Johnson. 2006. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice* 18, 2 (2006), 83–107.

[22] Ekwa Duala-Ekoko and Martin P. Robillard. 2011. Using Structure-Based Recommendations to Facilitate Discoverability in APIs. *Lecture Notes in Computer Science ECOOP 2011 – Object-Oriented Programming* (2011), 79–104. https://doi.org/10.1007/978-3-642-22655-7_5

[23] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. 2014. How do API documentation and static typing affect API usability? *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014* (2014).

[24] T. Espinha, A. Zaidman, and H. G. Gross. 2014. Web API growing pains: Stories from client developers and their code. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. 84–93.

[25] Giovanni Grano, Andrea Di Sorbo, Francesco Mercaldo, Corrado A. Visaggio, Gerardo Canfora, and Sebastiano Panichella. 2017. Android apps and user feedback: a dataset for software evolution and quality improvement. *Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics - WAMA 2017* (2017).

[26] J. Henkel and A. Diwan. [n. d.]. Catchup! capturing and replaying refactorings to support API evolution. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* ([n. d.]).

[27] R. Holmes and G.c. Murphy. [n. d.]. Using structural context to recommend source code examples. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* ([n. d.]).

[28] Daqing Hou and David M. Pletcher. 2011. An evaluation of the strategies of sorting, filtering, and grouping API methods for Code Completion. *2011 27th IEEE International Conference on Software Maintenance (ICSM)* (2011).

[29] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2014. Mining energy-greedy API usage patterns in Android apps: an empirical study. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014* (2014).

[30] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. API change and fault proneness: a threat to the success of Android apps. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013* (2013).

[31] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2014. How do API changes trigger stack overflow discussions? a study on the Android SDK. *Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014* (2014).

[32] Tyler Mcdonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. *2013 IEEE International Conference on Software Maintenance* (2013).

[33] Collin Mcmillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. 2012. Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications. *IEEE Transactions on Software Engineering* 38, 5 (2012), 1069–1087.

[34] Collin Mcmillan, Denys Poshyvanyk, and Mark Grechanik. 2010. Recommending source code examples via API call usages and documentation. *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering - RSSE 10* (2010).

[35] B. Morel and P. Alexander. 2004. SPARTACAS: automating component reuse and adaptation. *IEEE Transactions on Software Engineering* 30, 9 (2004), 587–600.

[36] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How Can I Use This Method? *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (2015).

[37] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. 2014. Automatic generation of release notes. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014* (2014).

[38] Laura Moreno and Andrian Marcus. 2017. Automatic software summarization: the state of the art. *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)* (2017).

[39] Evan Moritz, Mario Linares-Vasquez, Denys Poshyvanyk, Mark Grechanik, Collin Mcmillan, and Malcom Gethers. 2013. ExPort: Detecting and visualizing API usages in large source code repositories. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2013).

[40] Dan Morrill. 1970. Announcing the Android 1.0 SDK, release 1. (Jan 1970). https://android-developers.googleblog.com/2008/09/announcing-android-10-sdk-release-1.html

[41] Brad A. Myers and Jeffrey Stylos. 2016. Improving API usability. *Commun. ACM* 59, 6 (2016), 62–69.

[42] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. 2016. API code recommendation using statistical learning from fine-grained changes. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016* (2016).

[43] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. 2016. Mapping API Elements for Code Migration with Vector Representations. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. ACM, New York, NY, USA, 756–758.

[44] Marius Nita and David Notkin. 2010. Using Twinning to Adapt Programs to Alternative APIs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 205–214.

[45] Chris Parnin and Christoph Treude. 2011. Measuring API documentation on the web. *Proceeding of the 2nd international workshop on Web 2.0 for software engineering - Web2SE 11* (2011).

[46] Pujan Petersen, Stefan Hanenberg, and Romain Robbes. 2014. An empirical comparison of static and dynamic type systems on API usage in the presence of an IDE: Java vs. groovy with eclipse. *Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014* (2014).

[47] S. Raemaekers, A. van Deursen, and J. Visser. 2014. Semantic Versioning versus Breaking Changes: A Study of the Maven Repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. 215–224.

[48] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. *ACM SIGPLAN Notices* 49, 6 (May 2014), 419–428.

[49] Romain Robbes and Michele Lanza. 2010. Improving code completion with program history. *Automated Software Engineering* 17, 2 (Dec 2010), 181–212.

[50] Romain Robbes, Mircea Lungu, and David Röthlisberger. 2012. How do developers react to API deprecation? *Proceedings of the ACM SIGSOFT 20th International*

Symposium on the Foundations of Software Engineering - FSE 12 (2012).

[51] Martin P. Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurelio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vasquez, and et al. 2017. On-demand Developer Documentation. *2017 IEEE International Conference on Software Maintenance and Evolution* (2017).

[52] Anand Ashok Sawant and Alberto Bacchelli. 2015. A Dataset for API Usage. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories* (2015).

[53] Anand Ashok Sawant and Alberto Bacchelli. 2016. fine-GRAPE: fine-grained APi usage extractor – an approach and dataset to investigate API usage. *Empirical Software Engineering* 22, 3 (Apr 2016), 1348–1371.

[54] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. 2017. On the reaction to deprecation of clients of 4 1 popular Java APIs and the JDK. *Empirical Software Engineering* (2017).

[55] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API Documentation. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 643–652.

[56] Ferdian Thung, Shaowei Wang, David Lo, and Julia Lawall. 2013. Automatic recommendation of API methods from feature requests. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2013).

[57] Christoph Treude and Martin P. Robillard. 2016. Augmenting API documentation with insights from stack overflow. *Proceedings of the 38th International Conference on Software Engineering - ICSE 16* (2016).

[58] Pradeep K. Venkatesh, Shaohua Wang, Feng Zhang, Ying Zou, and Ahmed E. Hassan. 2016. What Do Client Developers Concern When Using Web APIs? An Empirical Study on Developer Forums and Stack Overflow. *2016 IEEE International Conference on Web Services (ICWS)* (2016).

[59] Roman Štrobl and Zdeněk Troníček. 2013. Migration from Deprecated API in Java. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, 38; Applications: Software for Humanity (SPLASH '13)*. ACM, New York, NY, USA, 85–86.

[60] Chenglong Wang, Jiajun Jiang, Jun Li, Yingfei Xiong, Xiangyu Luo, Lu Zhang, and Zhenjiang Hu. 2016. Transforming Programs between APIs with Many-to-Many Mappings. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 25:1–25:26. http://drops.dagstuhl.de/opus/volltexte/2016/6119

[61] Martin White, Christopher Vendome, Mario Linares-Vasquez, and Denys Poshyvanyk. 2015. Toward Deep Learning Software Repositories. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories* (2015).

[62] Deheng Ye, Zhenchang Xing, Chee Yong Foo, Jing Li, and Nachiket Kapre. 2016. Learning to Extract API Mentions from Informal Natural Language Discussions. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2016).

[63] Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. 2012. Automatic parameter recommendation for practical API usage. *2012 34th International Conference on Software Engineering (ICSE)* (2012).

[64] Jing Zhou and Robert J. Walker. 2016. API Deprecation: A Retrospective Analysis and Detection Method for Code Examples on the Web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 266–277.